NEC

User's Manual

RX4000

Real-Time Operating System

Fundamental

Target Device
VR4100 Series™

Document No. U13422EJ1V1UM00 (1st edition)
Date Published October 2001 N CP (K)

[MEMO]

Regional Information

Some information contained in this document may vary from country to country. Before using any NEC product in your application, please contact the NEC 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 Inc. (U.S.)
Santa Clara, California
Tel: 408-588-6000
800-366-9782
Fax: 408-588-6130
800-729-9288

NEC Electronics (Germany) GmbH
Duesseldorf, Germany
Tel: 0211-65 03 02
Fax: 0211-65 03 490

NEC Electronics (UK) Ltd.
Milton Keynes, UK
Tel: 01908-691-133
Fax: 01908-670-290

NEC Electronics Italiana s.r.l.
Milano, Italy
Tel: 02-66 75 41
Fax: 02-66 75 42 99

NEC Electronics (Germany) GmbH
Benelux Office
Eindhoven, The Netherlands
Tel: 040-2445845
Fax: 040-2444580

NEC Electronics (France) S.A.
Velizy-Villacoublay, France
Tel: 01-3067-5800
Fax: 01-3067-5899

NEC Electronics (France) S.A.
Madrid Office
Madrid, Spain
Tel: 091-504-2787
Fax: 091-504-2860

NEC Electronics (Germany) GmbH
Scandinavia Office
Taeby, Sweden
Tel: 08-63 80 820
Fax: 08-63 80 388

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 Singapore Pte. Ltd.
Novena Square, Singapore
Tel: 253-8311
Fax: 250-3583

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

NEC do Brasil S.A.
Electron Devices Division
Guarulhos-SP, Brasil
Tel: 11-6462-6810
Fax: 11-6462-6829

J01.2

[MEMO]

PREFACE

Readers	This manual is applicable to users engaged in the design or development of systems compatible with the VR4100 Series.
Purpose	The purpose of this manual is to provide an understanding to users of the RX4000 shown in the following configuration.
Organization	This manual is roughly organized from the following contents.

- Overview
- Nucleus
- Task Management Function
- Synchronous Communication Functions
- Interrupt Management Function
- Memory Pool Management Function
- Time Management Function
- Scheduler
- System Management Functions
- System Initialization
- Interface Library
- System Calls

How to Read This Manual Those who read this manual need to have a general knowledge of electricity, logic circuits, microcontrollers, the C language and assembly language.

To learn about the hardware functions or command functions of the VR4100 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

Prefix indicating power of 2 (Address space, memory capacity)

K (kilo) $2^{10} = 1024$

M (mega) $2^{20} = 1024^2$

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

○ Documents related to the VR4100 Series

Document Name	Document No.
VR4100™ User's Manual	U10050E
μPD30100 Data Sheet	U10428J Note
VR4102™ User's Manual	U12739E
μPD30102 Data Sheet	U12543E
VR4111™ User's Manual	U13137E
μPD30111 Data Sheet	U13211E

Note Available only in Japanese.

○ Documents related to development tools (User's Manuals)

Product Name		Document No.
RX4000 (Real-time OS)	Fundamental	This manual
	Technical	To be prepared
	Installation	To be prepared
RD4000 (Task Debugger)		To be prepared
AZ4000 (System Performance Analyzer)		To be prepared

TABLE OF CONTENTS

LIST OF FIGURES (1/2)

LIST OF FIGURES (2/2)

LIST OF TABLES (1/1)

[MEMO]

CHAPTER 1 OVERVIEW

Rapid advances in semiconductor technologies have led to the explosive spread of microprocessors such that they are now to be found in more fields than many would have imagined only a few years ago. In line with this spread, the number of processing programs that must be created for ever-newer high-performance, multi-function microprocessors is also increasing. This rule of growth makes it difficult to create processing programs specific to given hardware.

For this reason, there is a need for operating systems (OSs) that can fully exploit the capabilities of the latest generation of microprocessors.

Operating systems are broadly classified into two types: program-development OSs and control OSs. Program-development OSs are to be found in those environments in which standard OSs (e.g., MS-DOS™, Windows™, and UNIX™ OS) predominate because the hardware configuration to be used for development can be limited to some extent (e.g., personal computers).

Conversely, control OSs are incorporated into control units. That is, these OSs are found in those environments where standard OSs cannot easily be applied because the hardware configuration varies from system to system and because efficient operation matching the application is required.

At NEC, we developed and marketed the VR4100 series to offer a more powerful microprocessor, and on the other hand, in consideration of current market conditions, in order to adequately develop the functions of the new high-performance microprocessor, we developed and marketed RX4000.

RX4000 is a control OS for real-time, multitasking processing; it has been developed to increase the application range of high-performance, multi-function microprocessors and further improve their generality.

1.1 Overview

RX4000 is a built-in real-time, multitask control OS that provides a highly efficient real-time, multitasking environment to increase the application range of processor control units.

RX4000 is a high-speed, compact OS capable of being stored in and run from the ROM of a target system.

1.2 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 handling as a means of satisfying this demand. As control equipment has become more powerful, however, it has proved difficult for systems to satisfy these requirements by means of simple interrupt handling 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.

To overcome this problem, real-time operating systems have been designed.

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

1.3 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 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 throughput of the overall system through the parallel processing of multiple tasks.

1.4 Features

RX4000 has the following features:

(1) Conformity with μITRON3.0 specification

As a representative embedded type control OS architecture, RX4000 performs design which is compatible with μITRON3.0 specifications, and includes all the functions up to level S and all functions except Level E expanded synchronous communications.

The μITRON3.0 specification applies to a built-in, real-time control OS.

(2) High generality

RX4000 supports system-specific system calls, as well as those defined in the μITRON3.0 specification. RX4000 thus offers superior application system generality.

RX4000 can be used to create a real-time, multitask OS that is compact and optimum for the user’s needs because the functions (system calls) to be used by the application system can be selected at system construction.

(3) Realization of real-time processing and multitasking

RX4000 supports the following functions to realize complete real-time processing and multitasking:

- Task management function
- Task-associated handler function
- Task-associated synchronization function
- Synchronous communication function
- Interrupt management function
- Memory pool management function
- Time management function
- System management function
- Scheduling function

(4) Compact design

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.

(5) Application utility support

RX4000 supports the following utility to aid in system construction:

- System Configurator CF4000

(6) Cross tools

RX4000 supports the following cross tools for the V_R Series™:

- CodeWarrior™ (Metroworks Corporation)
- C Cross MIPE Compiler (Green Hills Software™, Inc.)

(7) RX830

RX4000 preserves compatibility with RX830 (μITRON Ver. 3.0).

1.5 Configuration

RX4000 consists of three subsystems: the nucleus, interface library, and configurator.
These subsystems are outlined below:

(1) Nucleus

The nucleus forms the heart of RX4000, a system that supports real-time, multitask control. The nucleus provides the following functions:

- Generation/initialization of a management object
- Processing of a system call issued by a processing program (task/non-task)
- Selection of the processing program (task/non-task) to be executed next, according to an event that occurs internal to or external to the target system

Management object generation/initialization and system call processing are executed by management modules. Processing program selection is performed by a scheduler.

(2) Interface library

When a processing program (task/non-task) is written in C, the external function format is used to issue a system call or call an extended SVC handler. The issue format that can be understood by the nucleus (nucleus issue format), however, differs from the external function format.

Therefore, the interface library is supported to translate a system call, issued in external function format or an extended SVC handler called in that format, into the nucleus issue format. The interface library thus acts as an agent between processing programs and the nucleus.

Furthermore, an interface library compatible with VR Series cross tools from Metroworks Corporation and Green Hills Software, Inc. are available with RX4000.

(3) Configurator CF4000

To construct a system using RX4000, information files containing the data to be supplied to RX4000 (system information table, branch table, and system information header file) are required.

As such information files consist of data arranged in a specified format, they can be written using an editor. However, files written in such a way are relatively difficult to write and subsequently understand.

RX4000 provides a utility for converting a file, created in a description format that offers much better writability and readability (CF definition file), to an information file.

This utility is Configurator CF4000. Configurator CF4000 takes a CF definition file, created in a specific format, as its input and outputs information files, such as system information tables, branch tables, or system information header files.

1.6 Applications

RX4000 is applicable for the following devices.

- Automobiles, robots and other control systems.
- Measuring instruments
- Exchangers, numerical controls, communication controls, plant controls and other control devices.
- Facsimile, copier and other OA machines.
- Medical treatment devices, space monitoring devices and other data collection and data calculation systems.

1.7 Execution Environment

This section explains the processing environment required by RX4000.

- **Processor**

  VR4100 Series

  μPD30100 (Other name: VR4100)

  μPD30102 (Other name: VR4102)

  μPD30111 (Other name: VR4111)

- **Peripheral hardware**
  RX4000 provides sample source files for that portion that is dependent on the hardware configuration of the execution environment (system initialization: boot processing and hardware initialization section).
  Therefore, simply rewriting the system initialization for each target system eliminates the need to use a specific peripheral hardware.

- **Memory requirements**
  The amount of memory required for RX4000 to execute processing is shown below.

Nucleus text section:	approximately 8 K to 18 Kbytes
Nucleus data section:	approximately 2 K to 3 Kbytes

  The maximum sizes shown above are applied when the system configuration specifies that the maximum priority range is set and that all the functions (system calls) provided by RX4000 are to be used. Therefore, by limiting the priority range or functions to be used, the required amount of memory can be reduced.

1.8 Development Environment

This section explains the hardware and software environments required to develop an application system.

(1) Hardware environment

(a) Host machine

- IBM-PC/AT™-compatible machine (Windows 95 base)
- PC-9800 Series (Windows 95 base)

(2) Software environment

(a) Cross Tools

- CodeWarrior (Metroworks Corporation)
- C Cross MIPE Compiler (Green Hills Software, Inc.)

(b) Debuggers

- PARTNER (Kyoto Microcomputer, Ltd.)
- MULTI™ (Green Hills Software Inc.)

(c) Task debugger

- RD4000 (NEC)

(d) System performance analyzer

- AZ4000 (NEC)

1.9 System Construction Procedure

System construction involves incorporating created load modules into a target system, using the file group copied from the RX4000 distribution media (floppy disk) to the user development environment (host machine).

The RX4000 system construction procedure is outlined below.

For details, refer to the **RX4000 User's Manual Installation**.

(1) Creating a CF definition file

(2) Creating an information definition file

- System information table (SIT)
- System call table (SCT)
- System information header file

Furthermore, these information tables are created using a configurator.

(3) Creating system initialization

- Boot processing
- Hardware initialization section
- Software initialization section

(4) Creating processing programs

- Task
- Task-associated handler
- Interrupt handler
- Cyclically activated handler
- Exception handler
- Extended SVC handler
- Interface library for an extended SVC handler

These programs are created using the C language or assembly language.

(5) Creating an initialization data save area (When CodeWarrior is used only)

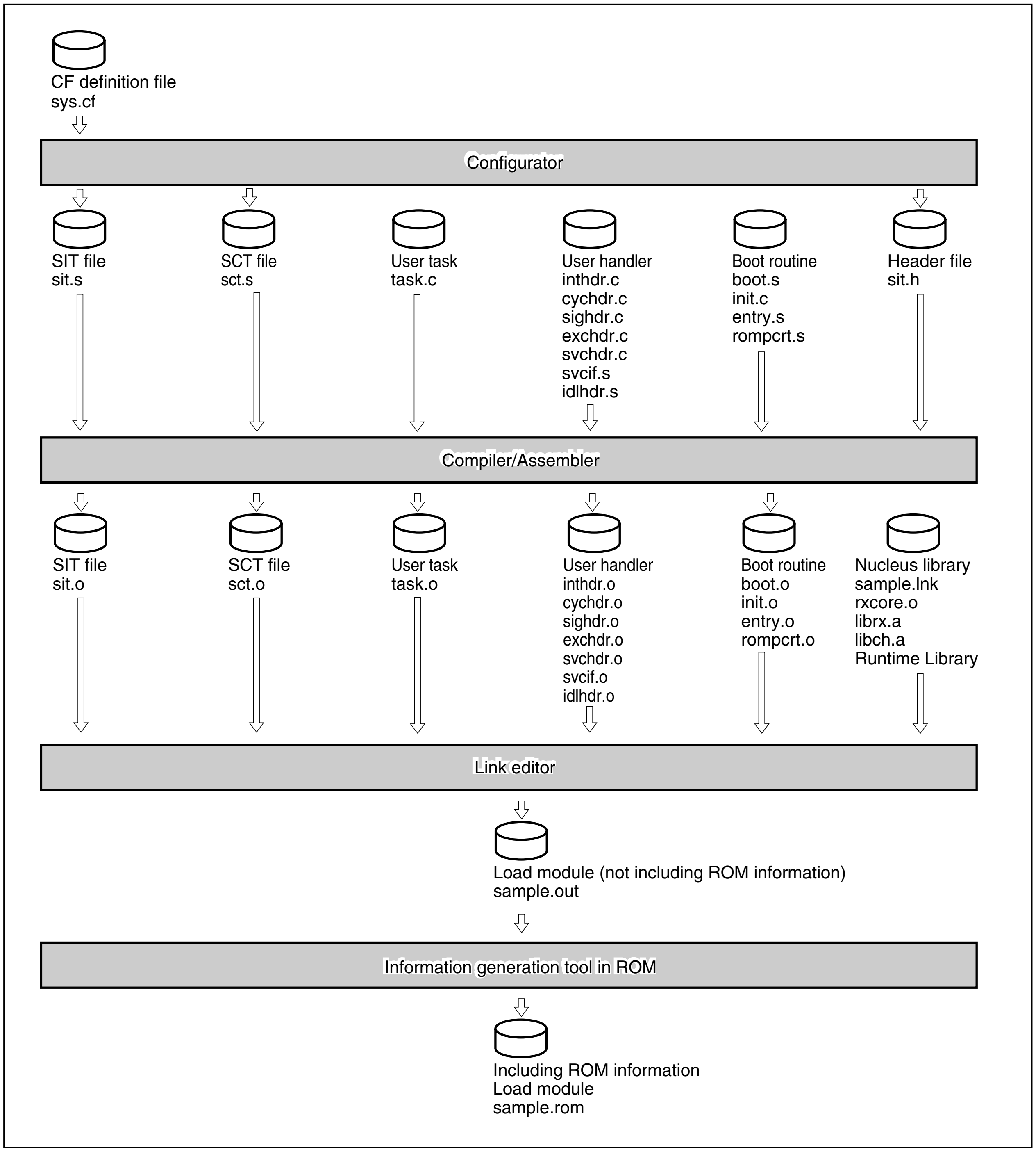
(6) Creating a link directive file

(7) Creating a load module

(8) Incorporating the load module into the system

An example of the system construction procedure when CodeWarrior is used is shown in Figure 1-1 and an example of the system construction procedure when the C Cross MIPE Compiler is used is shown in Figure 1-2.

Figure 1-1. System Construction Procedure (When CodeWarrior is used)



The files shown in Figure 1-1 are as follows:

- **CF definition file**

sys.cf:	CF definition file

- **SIT files**

sit.s:	System information table

- **SCT files**

sct.s:	Branch table

- **User task**

task.c:	Task

- **User handler**

inthdr.c:	Interrupt handler
cychdr.c:	Cyclically activated handler
sighdr.c:	Task-associated handler
exchdr.c:	Exception handler
svchdr.c:	Extended SVC handler
svcif.s:	Interface library for an extended SVC handler
idlhdr.s:	Idle handler

- **Boot routine**

boot.s:	Boot processing
init.c:	Hardware initialization section
entry.s:	Hardware initialization section (interrupt/exception entry)
rompcrt.s:	Initialization data save area

- **Header file**

sit.h:	System information header file

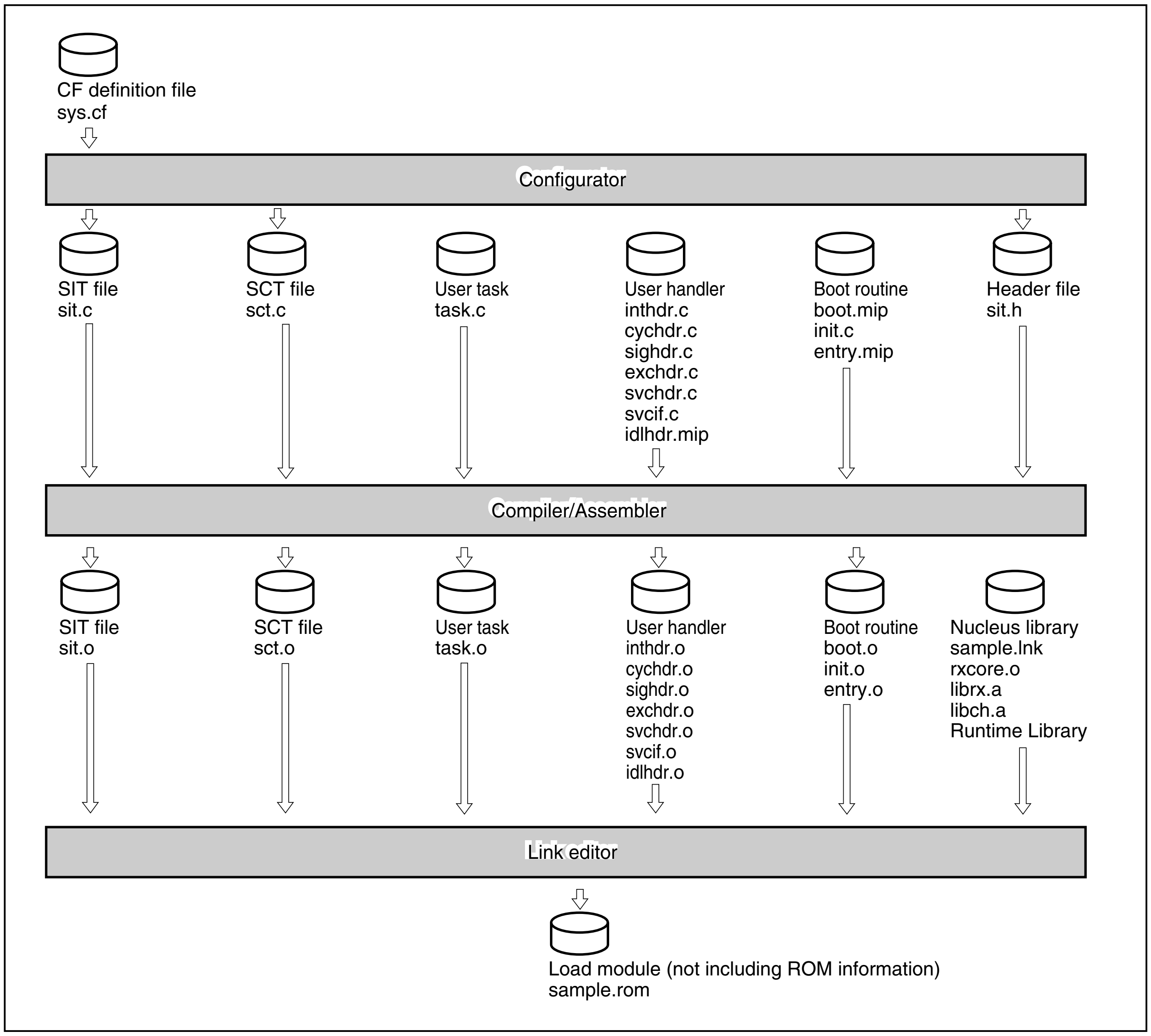
- **Nucleus library**

sample.lnk:	Link directive file
rxcore.o:	Nucleus common section
librx.a:	Nucleus library
libch.a:	Interface library for system calls

- **Load modules**

sample.out:	Not including ROM information
sample.rom:	Including ROM information

Figure 1-2. System Construction Procedure (When C Cross MIPE Compiler is used)



The files shown in Figure 1-2 are as follows:

- **CF definition file**

sys.cf:	CF definition file

- **SIT files**

sit.c:	System information table

- **SCT files**

sct.c:	Branch table

- **User task**

task.c:	Task

- **User handler**

inthdr.c:	Interrupt handler
cychdr.c:	Cyclically activated handler
sighdr.c:	Task-associated handler
exchdr.c:	Exception handler
svchdr.c:	Extended SVC handler
svcif.c:	Interface library for an extended SVC handler
idlhdr.mip:	Idle handler

- **Boot routine**

boot.mip:	Boot processing
init.c:	Hardware initialization section
entry.mip:	Hardware initialization section (interrupt/exception entry)

- **Header files**

sit.h:	System information header file

- **Nucleus library**

sample.lnk:	Link directive file
rxcore.o:	Nucleus common section
librx.a:	Nucleus library
libch.a:	Interface library for system calls

- **Load modules**

sample.rom:	Including ROM information

CHAPTER 2 NUCLEUS

This chapter explains concerning the nucleus which is the core of RX4000.

2.1 Overview

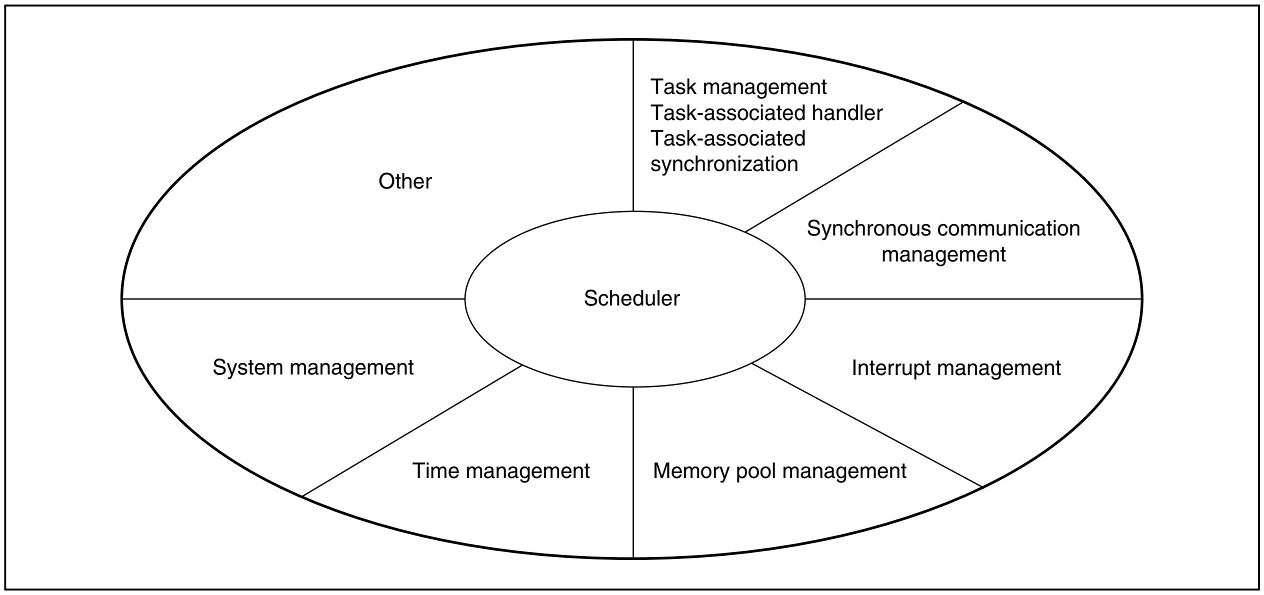
The nucleus forms the heart of RX4000, a system that supports real-time, multitask control. The nucleus provides the following functions:

- Generation/initialization of a management object
- Processing of a system call issued by a processing program (task/non-task)
- Selection of the processing program (task/non-task) to be executed next, according to an event that occurs internal to or external to the target system

Management object generation/initialization and system call processing are executed by management modules. Program selection is performed by a scheduler.

The configuration of the RX4000 nucleus is shown below.

Figure 2-1. Nucleus Configuration



2.2 Functions

The nucleus consists of a scheduler and various kinds of management modules.

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

See **CHAPTERS 3 TASK MANAGEMENT FUNCTION** through **8 SCHEDULER** for details of the individual functions.

(1) Task management function

This module manipulates and manages the states of a task, the minimum unit in which processing is performed by RX4000. For example, the module can generate, start, run, stop, terminate, and delete a task. The handler which accompanies the task is also managed.

(2) Synchronous communication function

This module enables three functions related to synchronous communication between tasks: exclusive control, wait, and communication.

Exclusive control function:	Semaphore
Wait function:	Event flag
Communication function:	Mailbox

(3) Interrupt management function

This module executes the processing related to a maskable interrupt, such as the registration of an indirectly activated interrupt mask, return from a directly activated interrupt handler, and change or acquisition of the interrupt-enabled level.

(4) Memory pool management function

This module manages the memory area specified at configuration, dividing it into the following two areas:

- RX4000 area
  Management objects
  Memory pool

- Processing program (task/non-task) area
  Text area
  Data area
  Stack area

RX4000 also applies dynamic memory pool management. For example, RX4000 provides a function for obtaining and returning a memory area to be used as a work area as required.

By exploiting this ability to dynamically manage memory, the user can utilize a limited memory area with maximum efficiency.

(5) Time management function

This module supports a timer operation function (such as delayed wake-up of a task or activation of a cyclically activated handler) that is based on clock interrupts generated by the software clock.

(6) Scheduler

By monitoring the dynamically changing states of tasks, this module manages and determines the order in which tasks are executed and optimally assigns tasks a processing time.

RX4000 determines the task execution order according to assigned priority levels and by applying the FCFS method. When started, the scheduler determines the priority levels assigned to the tasks, selects an optimum task from those ready to be executed (run or ready state), and optimally assigns tasks a processing time.

Remark In RX4000, the smaller the value of the priority assigned to the task, the higher the priority.

[MEMO]

CHAPTER 3 TASK MANAGEMENT FUNCTION

This chapter describes the task management function performed by RX4000.

3.1 Overview

Tasks are execution entities of arbitrary sizes, such that they are difficult to manage directly. RX4000 manages task states and tasks themselves by using management objects that correspond to tasks on a one-to-one basis.

Remark A task uses the execution environment information provided by a program counter, work registers, and the like when it executes processing. This information is called the task context. When the task execution is switched, the current task context is saved and the task context for the next task is loaded.

3.2 Task States

The task changes its state according to how resources required to execute the processing are obtained, whether an event occurs, and so on.

RX4000 classifies task states into the following seven types:

(1) non_existent state
A task in this state has not been generated or has been deleted.
A task in the non_existent state is not managed by RX4000 even if its execution entity is located in memory.

(2) dormant state
A task in this state has just been generated or has already completed its processing.
A task in the dormant state is not scheduled by RX4000.
This state differs from the wait state in the following points:

- All resources are released.
- The task context is initialized when the processing is resumed.
- A state manipulation system call causes an error.

(3) ready state
A task in this state is ready to perform its processing. This task has been waiting for a processing time to be assigned because another task having a higher (or the same) priority level is being performed.
A task in the ready state is scheduled by RX4000.

(4) run state

A task in this state has been assigned a processing time and is currently performing its processing.

Within the entire system, only a single task can be in the run state at any one time.

(5) wait state

A task in this state has been stopped because the requirements for performing its processing are not satisfied.

The processing of this task is resumed from the point at which it was stopped. As a task context required to resume the processing, the values that were being used immediately before the stop are restored.

RX4000 further divides tasks in the wait state into the following six groups, according to the conditions which caused the transition to the wait state:

Wake-up wait state:	A task enters this state if the counter for the task (registering the number of times the wake-up request has been issued) indicates 0x0 upon the issue of an slp_tsk or tslp_tsk system call.
Resource wait state:	A task enters this state if it cannot obtain a resource from the relevant semaphore upon the issue of a wai_sem or twai_sem system call.
Event flag wait state:	A task enters this state if a relevant event flag does not satisfy a predetermined condition upon the issue of a wai_flg or twai_flg system call.
Message wait state:	A task enters this state if it cannot receive a message from the relevant mailbox upon the issue of a rcv_msg or trcv_msg system call.
Memory block wait state:	A task enters this state if it cannot obtain a memory block from the relevant memory pool upon the issue of a get_blk or tget_blk system call.
Timeout wait state:	A task enters this state upon the issue of a dly_tsk system call.

(6) suspend state

A task in this state has been forcibly stopped by another task.

The processing of this task is resumed from the point at which it was stopped. As a task context required for resuming the processing, the values that were being used immediately before the stop are restored.

Remark RX4000 supports nesting of more than one level of the suspend state.

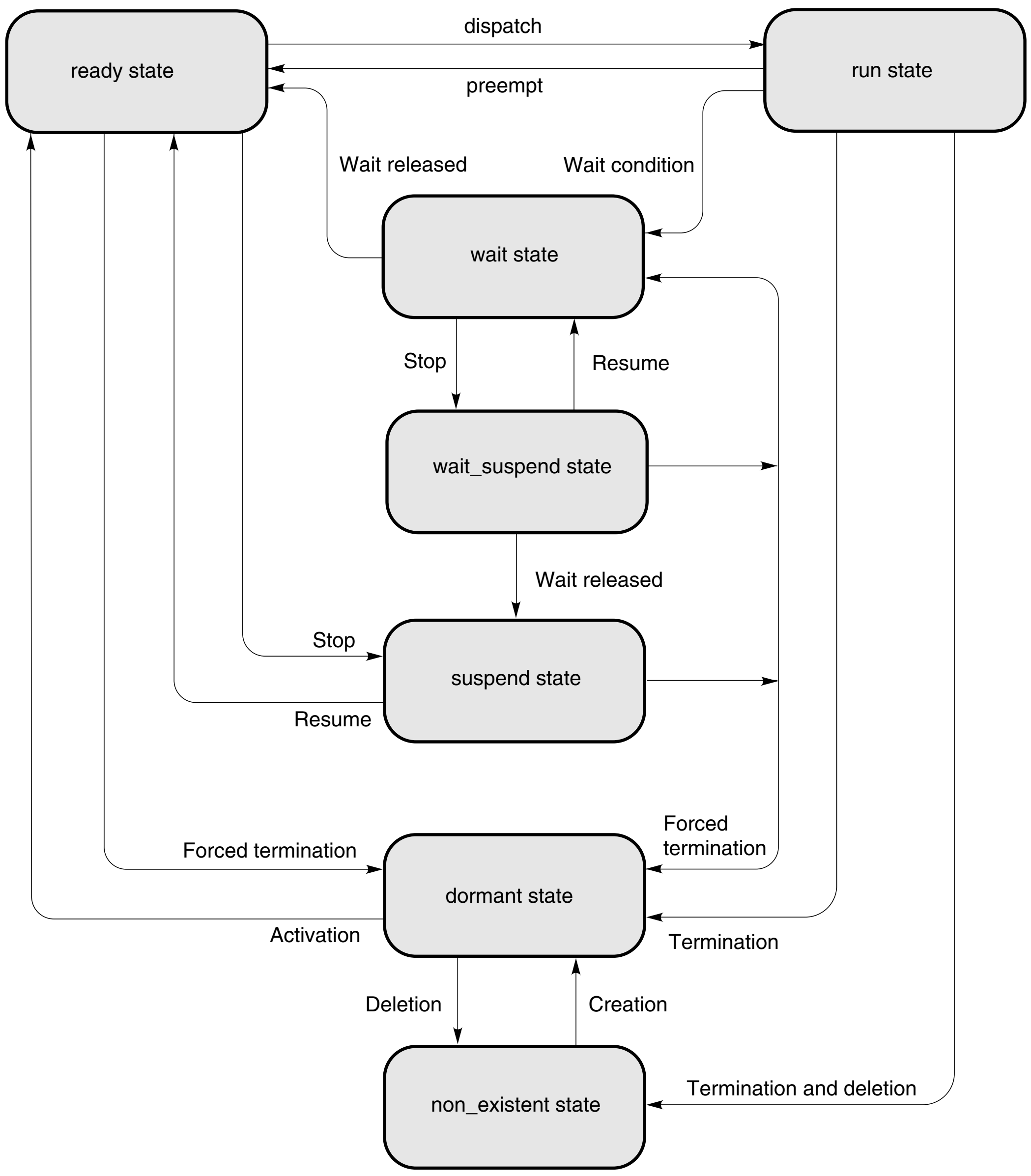
(7) wait_suspend state

This state is a combination of the wait and suspend states.

A task in this state has entered the wait state upon exiting from the suspend state, or has entered the suspend state upon exiting from the wait state.

Task status transitions are shown in Figure 3-1.

Figure 3-1. Task State Transition



3.3 Generating Tasks

To generate a task under RX4000, two types of interfaces are provided: A task is generated statically at system initialization (in the nucleus initialization section), or is generated dynamically according to a system call issued from a processing program.

Task generation under RX4000 consists of three steps: A task management area (management object) is allocated in system memory. Then, the allocated task management area is initialized. Finally, the task state is changed from the non_existent state to the dormant state.

(1) Static registration of a task

To register a task statically, specify that task during configuration.

RX4000 generates a task according to the information defined in the information files (system information table and system information header file) at system initialization, and makes the task manageable.

(2) Dynamic registration of a task

To register a task dynamically, issue a cre_tsk system call from a processing program (task).

RX4000 generates a task according to the information specified with parameters upon the issue of a cre_tsk system call, and makes the task manageable.

3.4 Activating Tasks

In task activation under RX4000, a task is switched from the dormant state to the ready state, and scheduled.

Furthermore, a task is activated by issuing a sta_tsk system call, specifying the task by the parameters.

3.5 Terminating Tasks

In task termination under RX4000, a task is switched from the ready state, run state, wait state, suspend state, or wait_suspend state to the dormant state and excluded from the schedule by RX4000.

Under RX4000, a task can be terminated in either of the following two ways:

Normal termination:	A task terminates upon completing all processing and when it need not be subsequently scheduled.
Forced termination:	When a number of troubles occur during processing and processing must be terminated immediately, this determines whether the task itself terminates or termination is accomplished from another task.

The task terminates upon the issue of the following system calls.

- ext_tsk system call
  A task which issued the ext_tsk system call is switched from the run state to the dormant state.

- exd_tsk system call
  A task which issued the exd_tsk system call is switched from the run state to the non_existent state.

- ter_tsk system call
  A task specified by the parameters is forcibly switched to the dormant state.

3.6 Deleting Tasks

In task deletion under RX4000, a task is switched from the run or dormant state to the non_existent state, and excluded from management by RX4000.

A task is deleted upon the issue of the following system calls.

- exd_tsk system call
  The task which issued the exd_tsk system call is switched from the run state to the non_existent state.

- del_tsk system call
  The task specified by the parameters is switched from the dormant state to the non_existent state.

3.7 Internal Processing of Task

RX4000 utilizes a unique means of scheduling to switch tasks.

Therefore, when describing a task's processing, please be careful of the following points.

(1) Saving/restoring the registers

When switching tasks, RX4000 saves and restores the contents of work registers in line with the function call conventions of VR series cross tool. This eliminates the need for coding processing to save the contents at the beginning of a task and that to restore the contents at the end.

If a task coded in assembly language uses a register for a register variable, however, the processing for saving the contents of that register must be coded at the beginning of the task, and that for restoring the contents at the end.

(2) Stack switching

When switching tasks, RX4000 switches to the special task stack of the selected task. The processing for switching the stack need not be coded at the beginning and end of the task.

(3) Limitations imposed on system calls

Some of the RX4000 system calls cannot be issued within a task.

The following system calls can be issued within a task:

- **Task management system calls**

cre_tsk	del_tsk	sta_tsk	ext_tsk	exd_tsk
ter_tsk	dis_dsp	ena_dsp	chg_pri	rot_rdq
rel_wai	get_tid	ref_tsk		

- **Task-associated handler function system calls.**

vdef_sig	vsnd_sig	vchg_sms	vref_sms

- **Task-associated synchronization system calls**

sus_tsk	rsm_tsk	frsm_tsk	slp_tsk	tslp_tsk
wup_tsk	can_wup			

- **Synchronous communication system calls**

cre_sem	del_sem	sig_sem	wai_sem	preq_sem
twai_sem	ref_sem	cre_flg	del_flg	set_flg
clr_flg	wai_flg	pol_flg	twai_flg	ref_flg
cre_mbx	del_mbx	snd_msg	rcv_msg	prcv_msg
trcv_msg	ref_mbx			

- **Interrupt management system calls**

def_int	loc_cpu	unl_cpu	dis_int	ena_int
chg_ims	reg_ims			

- **Memory pool management system calls**

cre_mpl	del_mpl	get_blk	pget_blk	tget_blk
rel_blk	ref_mpl			

- **Time management system calls**

set_tim	get_tim	dly_tsk	def_cyc	act_cyc
ref_cyc				

- **System management system calls**

get_ver	ref_sys	def_svc	viss_svc	def_exc

3.7.1 Acquiring task information

Task information is acquired upon the issue of a ref_tsk system call.

- ref_tsk system call
  Task information (such as extended information or the current priority) for the task specified by the parameters is acquired.
  The contents of the task information are as follows:

  - Extended information
  - Current priority
  - Task state
  - Type of the wait state
  - ID number of the object to be processed (semaphore, event flag, etc.)
  - Number of wake-up requests
  - Number of suspend requests

3.7.2 Task-associated handler

The task-associated handler is a task exclusive routine which performs processing of external phenomena (signals) generated by each task, and it positioned as an extension of the task which generated the phenomenon. For that reason, handler execution is done in the context of the object task. Also, the task-associated handler has the same priority order as the object task and is scheduled at the same level as the task.

3.7.3 Task-associated handler registration and canceling registration

Registration and canceling registration of a task-associated handler is accomplished by issuing the vdef_sig system call.

3.7.4 Return from the task-associated handler

Return processing from the task-associated handler is accomplished by issuing a return command at the end of the handler.

- return (INT retcd) Command
  The task-associated handler which is currently being run is terminated by the return processing method specified by the parameter. The following values can be specified in the parameter.

TRC_RET (0)	: Normal return
TRC_TSKEXT (–1)	: Return and terminate task normally

[MEMO]

CHAPTER 4 SYNCHRONOUS COMMUNICATION FUNCTIONS

This chapter describes the synchronous communication functions performed by RX4000.

4.1 Overview

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

Therefore, liaison functions between tasks are required, so that task execution will be suspended to await the result output by another task or until necessary conditions have been established to enable the processing to be continued.

In RX4000, these functions are called “synchronization functions.” The synchronization functions include a wait function and an exclusive control function. RX4000 provides semaphores that act as the exclusive control function and event flags that act as the wait function.

For multitasking, an inter-task communication function is also required to enable one task to receive the processing result from another.

In RX4000, this function is called a “communication function.” RX4000 provides mailboxes that act as the communication function.

4.2 Semaphores

Multitasking requires a function to prevent the resource contention which would occur when concurrently operating multiple tasks attempt to use a limited number of resources such as memory, coprocessors, files, and programs. To implement this contention preventive function, RX4000 provides non-negative counter-type semaphores.

The following system calls are used to dynamically manipulate a semaphore:

cre_sem	: Generates a semaphore
del_sem	: Deletes a semaphore
sig_sem	: Returns a resource
wai_sem	: Acquires a resource
preq_sem	: Acquires a resource (by polling)
twai_sem	: Acquires a resource (with timeout setting)
ref_sem	: Acquires semaphore information

Remark In RX4000, those elements required to execute tasks are called resources. In other words, resources comprehensively refer to hardware components such as processor, memory, and input/output equipment, as well as software components such as files and programs.

4.2.1 Generating semaphores

RX4000 provides two interfaces for generating semaphores. One enables the static generation of a semaphore during system initialization (in the nucleus initialization section). The other dynamically generates a semaphore by issuing a system call from within a processing program.

To generate a semaphore in RX4000, an area in system memory shall be allocated for managing that semaphore (as an object of management by RX4000), then initialized.

(1) Static registration of a semaphore

To statically register a semaphore, specify it during configuration.
RX4000 generates that semaphore according to the semaphore information defined in the information file (including system information tables and system information header subfiles) during system initialization. The semaphore is subsequently managed by RX4000.

(2) Dynamic registration of a semaphore

To dynamically register a semaphore, issue the cre_sem system call from within a processing program (task).
RX4000 generates that semaphore according to the information specified with parameters when the cre_sem system call is issued. The semaphore is subsequently managed by RX4000.

4.2.2 Deleting semaphores

A semaphore is deleted by issuing the del_sem system call.

- del_sem system call
  The del_sem system call deletes the semaphore specified with the parameter.
  That semaphore is then no longer managed by RX4000.
  If a task is queued into the queue of the semaphore specified by this system call parameter, that task shall be removed from the queue, after which it will leave the wait state (the resource wait state) and enter the ready state.
  E_DLT is returned to the task released from the wait state as the value returned in response to a system call (wai_sem or twai_sem) that triggered the transition of the task to the wait state.

4.2.3 Returning resources

A resource is returned by issuing the sig_sem system call.

- sig_sem system call
  By issuing the sig_sem system call, the task returns a resource to the semaphore specified by parameter (the semaphore counter is incremented by 0x1).
  If a task or tasks are queued into the queue of the semaphore specified by this system call parameter, the relevant resource is passed to the first task in the queue without being returned to the semaphore (thus, the semaphore counter is not incremented).
  Then, that task is removed from the queue, after which it leaves the wait state (the resource wait state) and enters the ready state. Or, it leaves the wait_suspend state and enters the suspend state.

4.2.4 Acquiring resources

A resource is acquired by issuing a wai_sem, preq_sem, or twai_sem system call.

- wai_sem system call

  By issuing a wai_sem system call, the task acquires a resource from the semaphore specified by a parameter (the semaphore counter is decremented by 0x1.)
  After issuing this system call, if the task cannot acquire the resource from the specified semaphore (no idle resource exists), the task itself is queued into the queue of this semaphore. Thus, the task leaves the run state and enters the wait state (the resource wait state).
  The resource wait state is canceled in the following cases, and the task returns to the ready state.

  - When a sig_sem system call is issued.
  - When a del_sem system call is issued and the specified semaphore is deleted.
  - When a rel_wai system call is issued and the wait state is forcibly canceled.
  - When a vsnd_sig system call is issued and the wait state is forcibly canceled.

Remark When a task queues in the wait queue of the specified semaphore, it is executed in the sequence (FIFO sequence or priority order sequence) specified when that semaphore was generated (during configuration or when a cre_sem system call was issued).

- preq_sem system call

  By issuing the preq_sem system call, the task acquires a resource from the semaphore specified by a parameter (the semaphore counter is decremented by 0x1.)
  After this system call is issued, if the task cannot acquire the resource from the specified semaphore (no idle resource exists), E_TMOUT is returned as the return value.

- twai_sem system call

  By issuing the twai_sem system call, the task acquires a resource from the semaphore specified by a parameter (the semaphore counter is decremented by 0x1.)
  After issuing this system call, if the task cannot acquire the resource from the specified semaphore (no idle resource exists), the task itself is queued into the queue of this semaphore. Thus, the task leaves the run state and enters the wait state (the resource wait state).
  The resource wait state is canceled in the following cases, and the task returns to the ready state.

  - When the given wait time specified by a parameter has elapsed.
  - When a sig_sem system call is issued.
  - When a del_sem system call is issued and the specified semaphore is deleted.
  - When a rel_wai system call is issued and the wait state is forcibly canceled.
  - When a vsnd_sig system call is issued and the wait state is forcibly canceled.

Remark When a task queues in the wait queue of the specified semaphore, it is executed in the sequence (FIFO sequence or priority order sequence) specified when that semaphore was generated (during configuration or when a cre_sem system call was issued).

4.2.5 Acquiring semaphore information

Semaphore information is acquired by issuing the ref_sem system call.

- ref_sem system call
  By issuing the ref_sem system call, the task acquires the semaphore information (extended information, queued tasks, etc.) for the semaphore specified by parameter.
  The semaphore information consists of the following:

  - Extended information
  - Whether tasks are queued
  - The number of currently available resources
  - The maximum number of resources specified when the semaphore was generated

4.2.6 Exclusive control using semaphores

The following is an example of using semaphores to manipulate the tasks under exclusive control.

Conditions

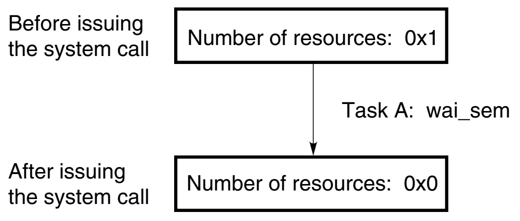
- Task priority
  Task A > Task B
- State of tasks
  Task A: run state
  Task B: ready state
- Semaphore attributes

Number of resources initially assigned to the semaphore:	0x1
Maximum number of resources that can be assigned to the semaphore:	0x5
Tasks queuing order:	FIFO

(1) Task A issues the wai_sem system call.

The number of resources assigned to this semaphore and managed by RX4000 is 0x1. Thus, RX4000 decrements the semaphore counter by 0x1.
At this time, task A does not enter the wait state (the resource wait state). Instead, it remains in the run state.
The relevant semaphore counter changes as shown in Figure 4-1.

Figure 4-1. State of the Semaphore Counter

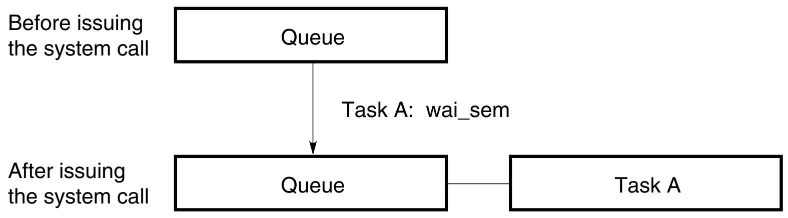


(2) Task A issues the wai_sem system call.

The number of resources assigned to this semaphore and managed by RX4000 is 0x0. Thus, RX4000 changes the state of task A from run to the wait state (resource wait state) and places the task at the end of the queue for this semaphore.

The queue of this semaphore changes as shown in Figure 4-2.

Figure 4-2. State of the Queue (When wai_sem is issued)



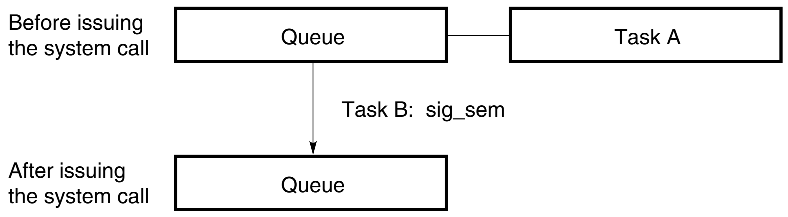
(3) As task A enters the resource wait state, the state of task B changes from ready to run.

(4) Task B issues the sig_sem system call.

At this time, the state of task A that has been placed in the queue of this semaphore changes from the resource wait state to ready state.

The queue of this semaphore changes as shown in Figure 4-3.

Figure 4-3. State of the Queue (When sig_sem is issued)

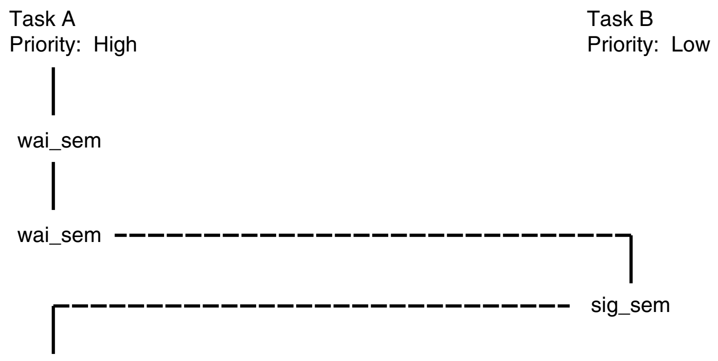


(5) The state of task A having the higher priority changes from ready to run.

At the same time, task B leaves the run state and enters the ready state.

Figure 4-4 shows the transition of exclusive control in steps (1) to (5).

Figure 4-4. Exclusive Control Using Semaphores



4.3 Event Flags

In multitask processing, an intertask wait function, in which other tasks wait to resume execution of processing until the results of processing by a given task are output, is necessary. In such a case, it is good to have a function for other tasks to judge whether or not the “processing results output” event has occurred or not, and in RX4000, an event flag is presented in order to realize this kind of function.

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

The following system calls regarding event flags are used to dynamically manipulate an event flag:

cre_flg	:	Generates an event flag.
del_flg	:	Deletes an event flag.
set_flg	:	Sets a bit pattern.
clr_flg	:	Clears a bit pattern.
wai_flg	:	Checks a bit pattern.
pol_flg	:	Checks a bit pattern (by polling).
twai_flg	:	Checks a bit pattern (with timeout setting).
ref_flg	:	Acquires event flag information.

4.3.1 Generating event flags

RX4000 provides two interfaces for generating event flags. One is for statically generating an event flag during system initialization (in the nucleus initialization section). The other is for dynamically generating an event flag by issuing a system call from within a processing program.

To generate an event flag in RX4000, an area in system memory shall be allocated for managing that event flag (as an object of management by RX4000), then initialized.

(1) Static registration of an event flag

To statically register an event flag, specify it during configuration.

RX4000 generates that event flag according to the event flag information defined in the information file (including system information tables and system information header subfiles) during system initialization. Subsequently, the event flag is managed by RX4000.

(2) Dynamic registration of an event flag

To dynamically register an event flag, issue the cre_flg system call from within a processing program (task).

RX4000 generates that event flag according to the information specified by a parameter when the cre_flg system call is issued. Subsequently, the event flag is managed by RX4000.

4.3.2 Deleting event flags

An event flag is deleted by issuing a del_flg system call.

- del_flg system call
  The del_flg system call deletes the event flag specified by a parameter.
  Then, that event flag is no longer managed by RX4000.
  If a task is queued into the queue of the event flag specified by this system call parameter, that task shall be removed from the queue, after which it will leave the wait state (the event flag wait state) and enter the ready state.
  E-DLT is returned to the task released from the wait state as the return value for the system call (wai_flg or twai_flg) that triggered the transition of the task to the wait state.

4.3.3 Setting a bit pattern

Setting of the event flag bit pattern is accomplished by issuing the set_flg system call.

- set_flg system call
  The set_flg system call sets a bit pattern for the event flag specified by a parameter.
  When this system call is issued, if the given condition for a task queued into the queue of the specified event flag is satisfied, that task shall be removed from the queue.
  Then, this task will leave the wait state (the event flag wait state) and enter the ready state. Or, it will leave the wait_suspend state and enter the suspend state.

4.3.4 Clearing a bit pattern

Clearing of the event flag bit pattern is accomplished by issuing the clr_flg system call.

- clr_flg system call
  The clr_flg system call clears the bit pattern of the event flag specified by a parameter.
  When this system call is issued, if the bit pattern of the specified event flag has already been cleared to zero, it is not regarded as an error. Pay particular attention to this point.

4.3.5 Checking a bit pattern

Checking of the event flag bit pattern is accomplished by issuing the wai_flg, pol_flg, or twai_flg system call.

- wai_flg system call
  The wai_flg system call checks whether the bit pattern is set to satisfy the wait condition required for the event flag specified by a parameter.
  If the bit pattern does not satisfy the wait condition required this point call is queued at the end of the queue of this event flag. Thus, the task leaves the run state and enters the wait state (the event flag wait state).
  The event flag wait state is canceled in the following cases, and the task returns to the ready state.

  - When a set_flg system call is issued and the required wait condition is set.
  - When a del_mbx system call is issued and this event flag is deleted.
  - When a rel_wai system call is issued and the wait state is forcibly canceled.
  - When a vsnd_sig system call is issued and the wait state is forcibly canceled.

- pol_flg system call

  The pol_flg system call checks whether the bit pattern is set to satisfy the wait condition required for the event flag specified by a parameter.

  If the bit pattern does not satisfy the wait condition required for the event flag specified by this system call parameter, E_TMOUT is returned as the return value.

- twai_flg system call

  The twai_flg system call checks whether the bit pattern is set to satisfy the wait condition required for the event flag specified by a parameter.

  If the bit pattern does not satisfy the wait condition required for the event flag specified by this system call parameter, the task that issues this system call is queued at the end of the queue for this event flag. Thus, the task leaves the run state and enters the wait state (the event flag wait state).

  The event flag wait state is canceled in the following cases, and the task returns to the ready state.

  - Once the given wait time specified by parameter has elapsed.
  - When a set_flg system call is issued and the required wait condition is set.
  - When a del_mbx system call is issued and this event flag is deleted.
  - When a rel_wai system call is issued and the wait state is forcibly canceled.
  - When a vsnd_sig system call is issued and the wait state is forcibly canceled.

Also, the event flag wait conditions and processing when the conditions are established can be specified as follows in RX4000.

(1) Wait conditions

- AND wait

  The wait state continues until all bits to be set to 1 in the required bit pattern have been set in the relevant event flag.
- OR wait

  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.

(2) When the condition is satisfied

- Clearing a bit pattern

  When the wait condition specified for the event flag is satisfied, the bit pattern for the event flag is cleared.

4.3.6 Acquiring event flag information

Event flag information is acquired by issuing the ref_flg system call.

- ref_flg system call

  By issuing the ref_flg system call, the task acquires the event flag information (extended information, queued tasks, etc.) for the event flag specified by a parameter.

  Details of event flag information are as follows:

  - Extended information
  - Whether tasks are queued
  - Current bit pattern

4.3.7 Wait function using event flags

The following is an example of manipulating the tasks under wait and control using event flags.

Conditions

- Task priority

  Task A > Task B
- State of tasks

  Task A : run state

  Task B : ready state
- Event flag attributes

  Initial bit pattern : 0x0

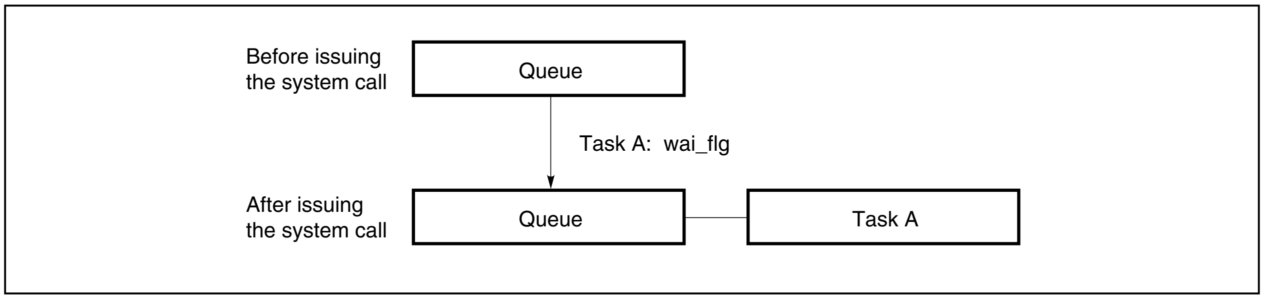
  The number of tasks that can be placed in the queue : One task

(1) Task A issues the wai_flg system call. The required bit pattern is 0x1 and the wait condition is TWF_ANDW|TWF_CLR.

The current bit pattern of the relevant event flag managed by RX4000 is 0x0. Thus, RX4000 changes the state of task A from run to wait (the event flag wait state). Then, task A is queued at the end of the queue for this event flag.

The queue of this event flag changes as shown in Figure 4-5.

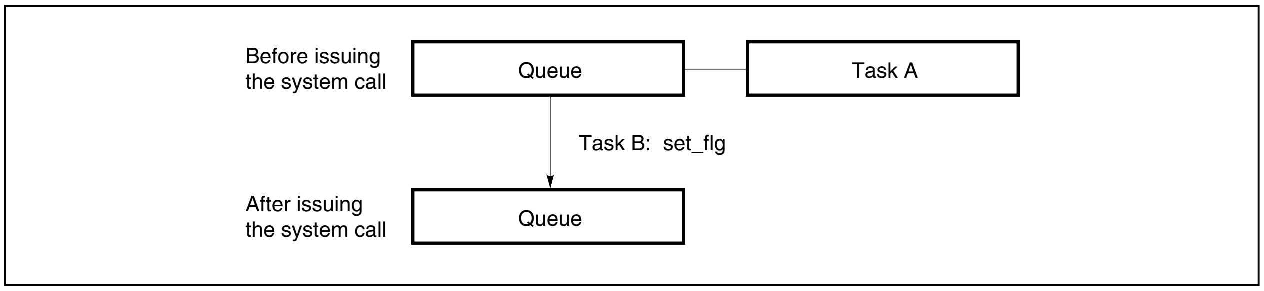
Figure 4-5. State of the Queue (When wai_flg is issued)



(2) As task A enters the event flag wait state, the state of task B changes from ready to run.

(3) Task B issues the set_flg system call. The bit pattern is set to 0x1.
This bit pattern satisfies the wait condition for task A that has been queued into the queue of the relevant event queue. Thus, task A leaves the event flag wait state and enters the ready state.
Since TWF_CLR was specified when task A issued the wai_flag system call, the bit pattern of this event flag is cleared.
The queue for this event flag changes as shown in Figure 4-6.

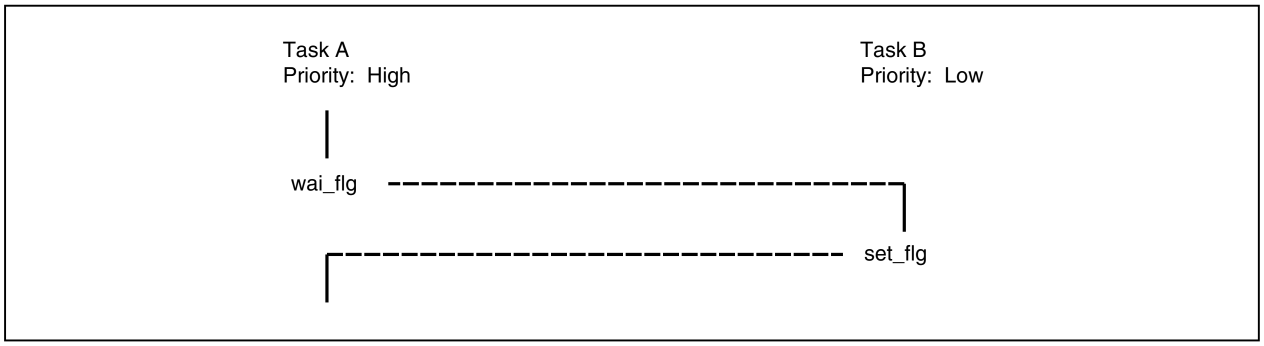
Figure 4-6. State of the Queue (When set_flg is issued)



(4) The state of task A having the higher priority changes from ready to run.
At the same time, task B leaves the run state and enters the ready state.

Figure 4-7 shows the transition of wait and control by event flags in steps (1) to (4).

Figure 4-7. Wait and Control by Event Flags



4.4 Mailboxes

Multitasking requires an inter-task communication function, so that the tasks can be informed of the results output by other tasks. To implement this function, RX4000 provides mailboxes.

The mailboxes used in RX4000 have two different queues, one dedicated to tasks and the other dedicated to messages. They can be used for both an inter-task message communication function and an inter-task wait function.

The following mailbox-related system calls are used to dynamically operate a mailbox.

cre_mbx	:	Generates a mailbox.
del_mbx	:	Deletes a mailbox.
snd_msg	:	Sends a message.
rcv_msg	:	Receives a message.
prcv_msg	:	Receives a message (by polling).
trcv_msg	:	Receives a message (with timeout setting).
ref_mbx	:	Acquires mailbox information

4.4.1 Generating mailboxes

RX4000 provides two interfaces for generating mailboxes. One is for statically generating a mailbox during system initialization (in the nucleus initialization section). The other is for dynamically generating a mailbox by issuing a system call from within a processing program.

To generate a mailbox in RX4000, an area in system memory shall be allocated for managing that mailbox (as an object of management by RX4000), then initialized.

(1) Static registration of a mailbox

To statically register a mailbox, specify it during configuration.

RX4000 generates the mailbox according to the mailbox information defined in the information file (including system information tables and system information header subfiles) during system initialization. Subsequently, the mailbox is managed by RX4000.

(2) Dynamic registration of a mailbox

To dynamically register a mailbox, issue the cre_mbx system call from within a processing program (task).

RX4000 generates the mailbox according to the information specified by a parameter when the cre_mbx system call is issued. Subsequently, the mailbox is managed by RX4000.

4.4.2 Deleting mailboxes

A mailbox is deleted by issuing the del_mbx system call.

- del_mbx system call

  The del_mbx system call deletes the mailbox specified by a parameter.

  Then, that mailbox is no longer managed by RX4000.

  If a task is queued into the task queue of the mailbox specified by this system call parameter, that task shall be removed from the task queue, after which it will leave the wait state (the message wait state) and enter the ready state.

  E-DLT is returned to the task released from the wait state as the return value for the system call (rcv_msg or trcv_msg) that triggered the transition of the task to the wait state.

4.4.3 Sending a message

A message is sent from a task by issuing a snd_msg system call.

- snd_msg system call

  Upon the issue of a snd_msg system call, the task transmits a message to the mailbox specified by a parameter.

  If a task or tasks are queued into the task queue of the mailbox specified by this system call parameter, the message is delivered to the first task in the task queue without being queued into the mailbox.

  Then, the first task is removed from the queue, after which it leaves the wait state (the message wait state) and enters the ready state. Or, it leaves the wait_suspend state and enters the suspend state.

  If no tasks are queued in the task wait queue of the object mail box, the message is placed in the message wait queue of the object mail box.

Remark Queuing of a message into the message wait queue of the mailbox specified by the system call parameter is performed in the order (FIFO or according to priority) specified when the mailbox was generated (when configuring or when the cre_mbx system call is issued).

4.4.4 Receiving a message

A message is received by the task upon the issue of the rcv_msg, prcv_msg, or trcv_msg system call.

- rcv_msg system call
  Upon the issue of a rcv_msg system call, the task receives a message from the mailbox specified by a parameter.
  If the task cannot receive a message from the mailbox specified by this system call parameter (no message exists in the message queue of that mailbox), the task that issued this system call is queued at the end of the task queue for this mailbox. Thus, the task leaves the run state and enters the wait state (the message wait state).
  The message wait state is canceled in the following cases and the task returns to the ready state.

  - When a snd_msg system call is issued.
  - When a del_mbx system call is issued and this mailbox is deleted.
  - When a rel_wai system call is issued and the wait state is forcibly canceled.
  - When a vsnd_sig system call is issued and the wait state is forcibly canceled.

Remark When a task queues in the task wait queue of the specified mailbox, it is executed in the sequence (FIFO sequence or priority order sequence) specified when that mailbox was generated (during configuration or when a cre_mbx system call was issued).

- prcv_msg system call
  Upon the issue of the prcv_msg system call, the task receives a message from the mailbox specified by a parameter.
  If the task cannot receive a message from the mailbox specified by this system call parameter (no message exists in the message queue for that mailbox), E_TMOUT is returned as the return value.

- trcv_msg system call
  Upon the issue of the trcv_msg system call, the task receives a message from the mailbox specified by a parameter.
  If the task cannot receive a message from the mailbox specified by this system call parameter (no message exists in the message queue for that mailbox), the task that issued this system call is queued at the end of the task queue for this mailbox. Thus, the task leaves the run state and enters the wait state (the message wait state).
  The message wait state is canceled in the following cases and the task returns to the ready state.

  - When the given time specified by parameter has elapsed.
  - When a snd_msg system call is issued.
  - When a del_mbx system call is issued and this mailbox is deleted.
  - When a rel_wai system call is issued and the wait state is forcibly canceled.
  - When a vsnd_sig system call is issued and the wait state is forcibly canceled.

Remark When a task queues in the task wait queue of the specified mailbox, it is executed in the sequence (FIFO sequence or priority order sequence) specified when that mailbox was generated (during configuration or when a cre_mbx system call was issued).

4.4.5 Messages

Under RX4000, all items of information exchanged between tasks, via mailboxes, are called "messages."

Messages can be transmitted to an arbitrary task via a mailbox. In inter-task communication under RX4000, however, only the start address of a message is delivered to a receiving task, enabling the task to access the message. The contents of the message are not copied to any other area.

(1) Allocating message areas

NEC recommends that the memory pool managed by RX4000 be allocated for messages. To make a memory pool area available for a message, the task should issue a get_blk, pget_blk, or tget_blk system call.

The first four bytes of each message are used as the block for linkage to the message queue when queued. Therefore, if areas other than the memory pool are allocated for messages, these message areas must be aligned with a 4-byte boundary.

(2) Composition of messages

RX4000 does not prescribe the length and composition of messages to be transmitted to mailboxes. The message length, except for the first four bytes, and its composition shall be defined by the tasks that communicate with each other via mailboxes.

Caution RX4000 prescribes that the first four bytes of each message are used as the block for linkage to the message queue when queued. For this reason, when a message is transmitted to the relevant mailbox, the first four bytes of the message must be set to 0x0 before the snd_msg system call is issued.

If the first four bytes of the message are set to a value other than 0x0 when the snd_msg system call is issued, RX4000 determines that this message has already been queued into the message queue. Thus, RX4000 does not send the message to the mailbox and returns E_OBJ as the return value.

4.4.6 Acquiring mailbox information

Mailbox information is acquired by issuing a ref_mbx system call.

- ref_mbx system call

  Upon the issue of a ref_mbx system call, the task acquires the mailbox information (extended information, queued tasks, etc.) for the mailbox specified by a parameter.

  The mailbox information consists of the following:

  - Extended information
  - Whether tasks are queued
  - Whether messages are queued

4.4.7 Inter-task communication using mailboxes

The following is an example of manipulating the tasks under inter-task communication using mailboxes.

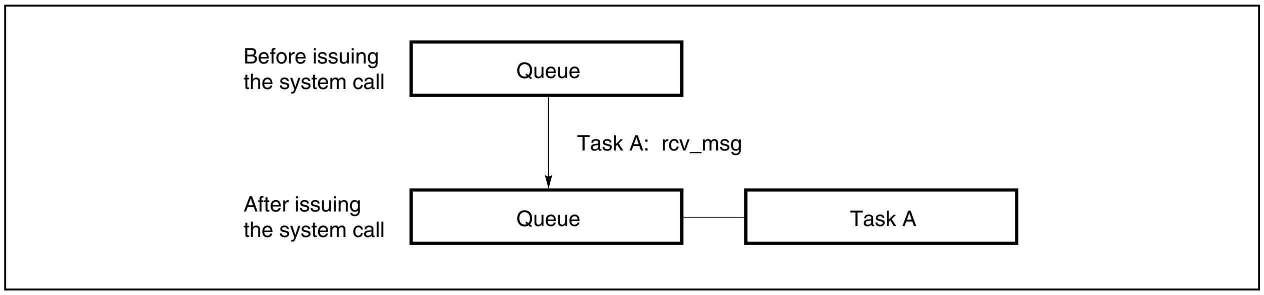
Conditions

- Task priority
  Task A > Task B
- State of tasks
  Task A: run state
  Task B: ready state
- Mailbox attributes

Task queuing order:	FIFO
Message queuing order:	FIFO

(1) Task A issues a rcv_msg system call.
No message is queued into the message queue of the relevant mailbox managed by RX4000. Thus, RX4000 changes the state of task A from run to wait (the message wait state). The task is queued at the end of the task queue for this mailbox.
The task queue for this mailbox changes as shown in Figure 4-8.

Figure 4-8. State of Task Queue (When the rcv_msg is issued)



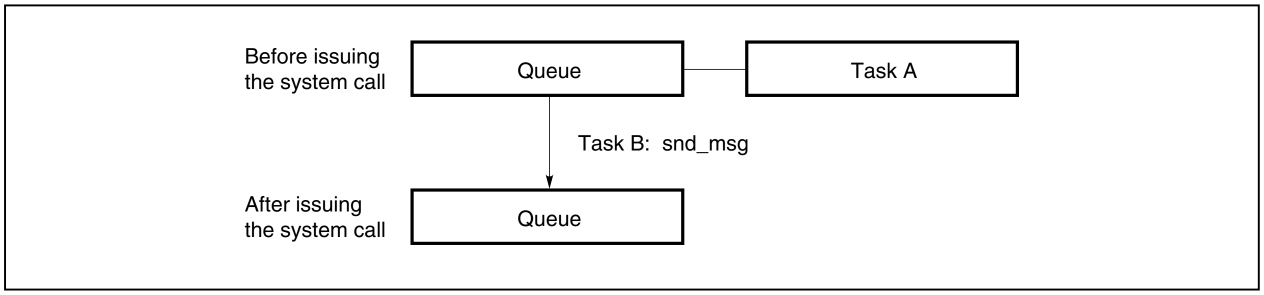
(2) As task A enters the message wait state, the state of task B changes from ready to run.

(3) Task B issues the get_blk system call.
By means of this system call, a memory pool area is allocated for a message (as a memory block).

(4) Task B writes a message into this memory block.

(5) Task B issues the snd_msg system call.
This changes the state of task A that has been placed in the task wait for the relevant mailbox from the message wait state to ready state.
The task queue for this mailbox changes as shown in Figure 4-9.

Figure 4-9. State of Task Queue (When the snd_msg is issued)

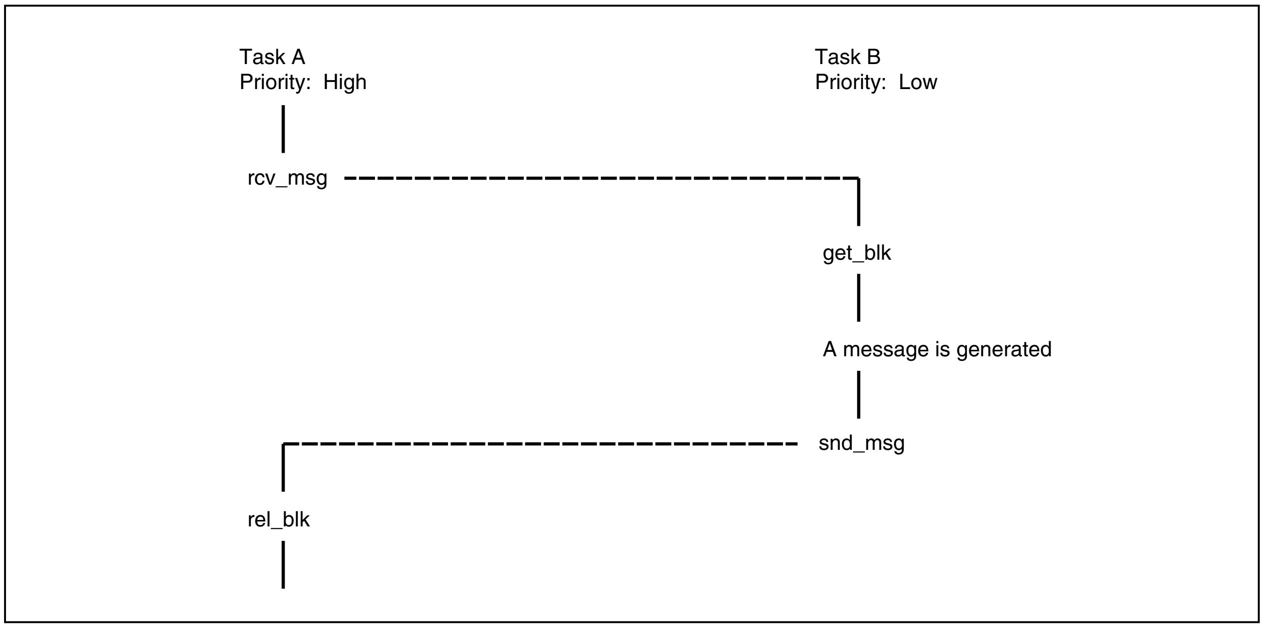


(6) The state of task A having the higher priority changes from ready to run.
At the same time, task B leaves the run state and enters the ready state.

(7) Task A issues the rel_blk system call
This releases the memory block allocated for the message in the memory pool.

The flow of communications between tasks as explained in (1) to (7) is shown in Figure 4-10.

Figure 4-10. Inter-Task Communication Using Mailboxes



CHAPTER 5 INTERRUPT MANAGEMENT FUNCTION

This chapter describes the interrupt management function provided by RX4000.

5.1 Overview

The RX4000 interrupt management function enables the following:

- Management of an interrupt handler
- Setting the interrupt mask
- Return from an interrupt handler
- Processing the actual interrupt

5.2 Interrupt Handler

An interrupt handler is a routine dedicated to interrupt processing. Upon the occurrence of an interrupt, the interrupt handler is initiated immediately and handled independently of all other tasks. Therefore, if a task having the highest priority in the system is being executed upon the occurrence of an interrupt, its processing is suspended and control is passed to the interrupt handler.

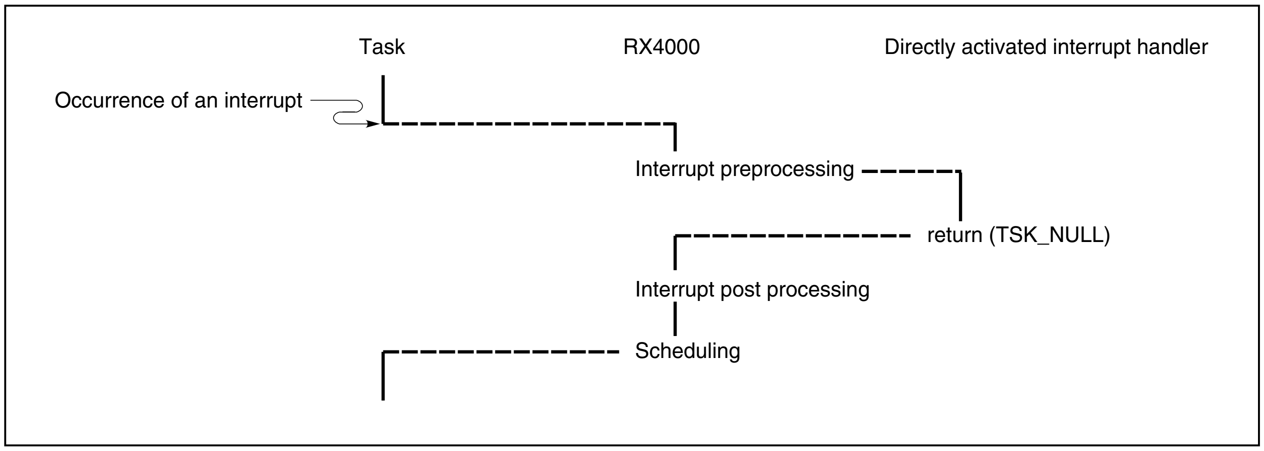
The interrupt handler is started after interrupt preprocessing by RX4000 (saving register, stack switching, etc.) is executed.

If a system call is issued while the interrupt handler is performing processing, scheduling is performed in a way specific to RX4000.

That is, if a system call (chg_pri, sig_sem, etc.) that requires task scheduling is issued during processing by the interrupt handler, RX4000 merely queues the tasks into the queue. The actual processing of task scheduling is batched and deferred until a return from the interrupt handler has been made (by issuing an ret_int system call and return instruction).

The flow of the interrupt handler's operation is shown in Figure 5-1.

Figure 5-1. Flow of Processing Performed by Directly Activated Interrupt Handler


Task
RX4000
Directly activated interrupt handler
Occurrence of an interrupt
Interrupt preprocessing
return (TSK_NULL)
Interrupt post processing
Scheduling

5.2.1 Registering interrupt handler

RX4000 provides two interfaces for registering an interrupt handler. One involves registering it statically during system initialization (in the nucleus initialization section). The other involves registering it dynamically by issuing a system call from within a processing program.

To register an interrupt handler with RX4000, an area in system memory shall be allocated for managing the interrupt handler (as an object of RX4000 management), then initialized.

(1) Static registration of an interrupt handler

To register an interrupt handler statically, specify it during configuration.

RX4000 performs the processing required to register the interrupt handler, based on the relevant information defined in the information file (including system information tables and system information header subfiles) during system initialization. The interrupt handler is subsequently managed by RX4000.

(2) Dynamic registration of an interrupt handler

To dynamically register an interrupt handler, issue the def_int system call from within a processing program (task or non-task).

RX4000 performs the processing required to register the interrupt handler, according to the information specified by the parameters when the def_int system call is issued. The interrupt handler is subsequently managed by RX4000.

5.2.2 Internal processing performed by the interrupt handler

When describing the processing to be performed by the interrupt handler, note the following:

(1) Saving/restoring the registers

Based on the function call protocol for the cross tools for the VR Series, RX4000 saves the work registers and restores them. Therefore, it is not necessary to describe processing related to save/restoration of the work registers.

Caution RX4000 does not switch the gp register when control is passed to the interrupt handler.

(2) Stack switching

RX4000 performs stack switching when control is passed to the interrupt handler and also upon return from the interrupt handler. Therefore, the interrupt handler does not have to switch to the interrupt handler stack when it starts, nor switch to the original stack upon the completion of its processing. Stack switching should not, therefore, be described in the coding for the interrupt handler.

If the interrupt handler stack is not defined during configuration, however, stack switching is not performed by RX4000. In this case, the system continues to use the stack being used upon the occurrence of the interrupt.

(3) Limitations imposed on system calls

Some of the system calls in RX4000 cannot be issued within the interrupt handler.
The following lists the system calls that can be issued during the processing of an interrupt handler:

- Task management system calls
  sta_tsk chg_pri rot_rdq rel_wai get_tid
  rer_tsk
- Task-associated handler function system call
  vsnd_sig
- Task-associated synchronization system calls
  sus_tsk rsm_tsk frsm_tsk wup_tsk can_wup
- Synchronous communication system calls
  sig_sem preq_sem ref_sem set_flg clr_flg
  pol_flg ref_flg snd_msg prcv_msg ref_mbx
- Interrupt management system calls
  def_int ret_int ret_wup dis_int ena_int
  chg_ims ref_ims
- Memory pool management system calls
  pget_blk rel_blk ref_mpl
- System management system calls
  get_ver ref_sys def_svc viss_svc def_exc

(4) Return processing from the interrupt handler

Return processing from the interrupt handler is performed by issuing a return instruction upon the completion of interrupt handler operation.

- return (TSK_NULL) instruction
  Performs return from the indirectly activated interrupt handler.

- return (ID *tskid*) instruction
  Issues a wake-up request to the task specified by the parameters, then returns from the indirectly activated interrupt handler.

When a system call (chg_pri, sig_sem, etc.) that requires task scheduling is issued during processing by an interrupt handler, RX4000 merely queues the tasks into the queue. The actual processing of task scheduling is batched and deferred until return from the interrupt handler has been made (by issuing a return instruction).

Caution The return instruction does not notify the external interrupt controllers that operation of the interrupt handler has terminated (the EOI command is not issued). Therefore, if a return is made from an interrupt handler that was initiated by an external interrupt request, notification of the termination of interrupt handler operation must be posted to the relevant interrupt controller before these system calls are issued.

5.3 Disabling/Enabling Maskable Interrupt Acceptance

RX4000 provides a function for disabling or enabling the acceptance of maskable interrupts, so that whether maskable interrupts are accepted can be specified from a user processing program.

This function is used by issuing the following system calls from within a task or interrupt handler.

- loc_cpu system call
  The loc_cpu system call disables the acceptance of maskable interrupts, as well as the performing of dispatch processing (task scheduling).
  Once this system call has been issued, control is not passed to any other task or interrupt handler until the unl_cpu system call is issued.

- unl_cpu system call
  The issue of the unl_cpu system call enables the acceptance of maskable interrupts, and resuming dispatch processing (task scheduling).

- dis_int system call
  Disables the acceptance of a maskable interrupt.

- ena_int system call
  Enables the acceptance of a maskable interrupt.

Figure 5-2 shows the flow of control if an interrupt is not masked and Figure 5-3 shows the flow of control if the loc_cpu system call is issued.

Figure 5-2. Flow of Control if Interrupt Mask Processing is Not Performed

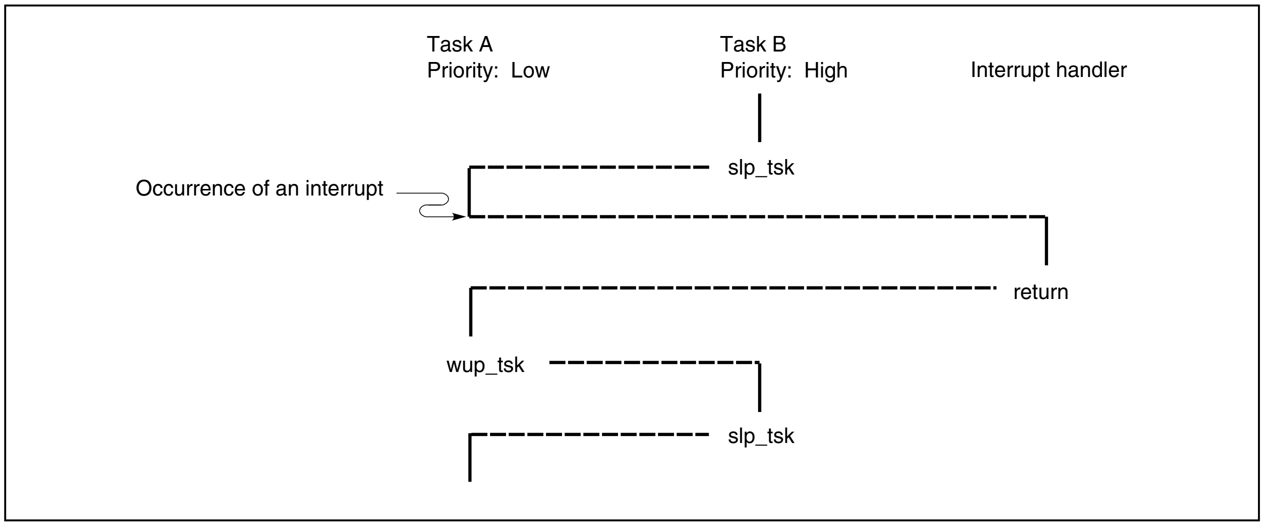
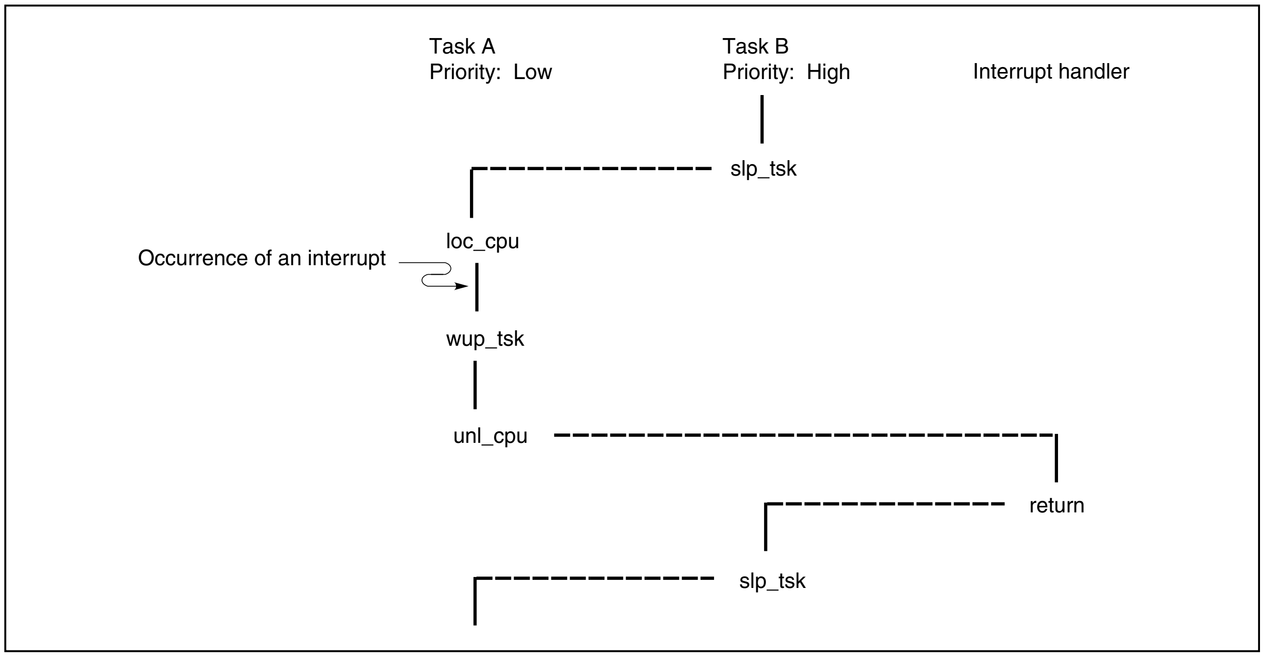


Figure 5-3. Flow of Control if the loc_cpu System Call is Issued



5.4 Changing/Acquiring Interrupt Mask

The interrupt mask setting is changed and acquired using chg_ims or acquired using ref_ims.

- chg_ims system call
  Upon the issue of a chg_ims system call, the interrupt mask setting of the processor is changed to the value specified by the parameter.

- ref_ims system call
  Upon the issue of an ref_ims system call, the task acquires the interrupt mask setting of the processor.

5.5 Nonmaskable Interrupts

A nonmaskable interrupt is not subject to management based on interrupt priority and has priority over all other interrupts. It can be accepted even if the processor is placed in the interrupt disabled state (clearing IE bit of status register).

Therefore, even while processing is being executed by RX4000 or an interrupt handler, a non-maskable interrupt can be accepted.

If a system call is issued during the processing of an interrupt handler that supports non-maskable interrupts, its operation cannot be assured under RX4000.

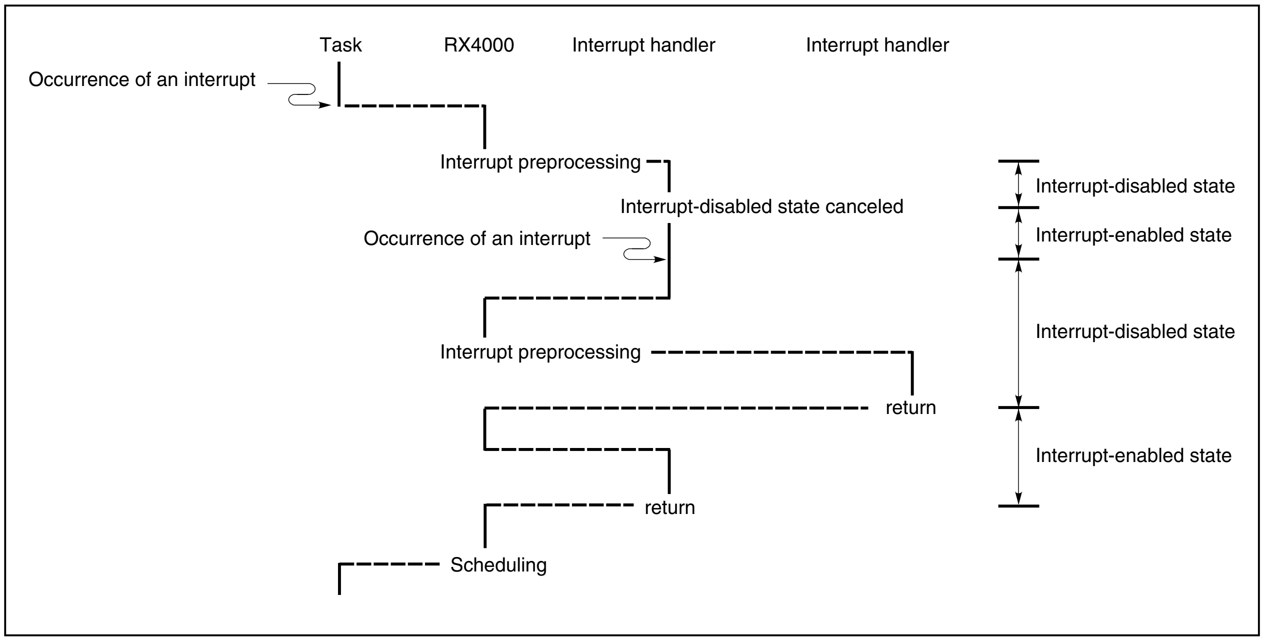
5.6 Multiple Interrupts

The occurrence of another interrupt while processing is being performed by an interrupt handler is called “multiple interrupts.” RX4000 also responds to multiple interrupt.

All interrupt handlers, however, start their operation in the interrupt-disabled state (clearing IE bit of status register). To enable the acceptance of multiple interrupts, the canceling of the interrupt disabled state should be described in the interrupt handler.

Figure 5-4 shows the flow of the processing for handling multiple interrupts.

Figure 5-4. Flow of Processing for Handling Multiple Interrupts



CHAPTER 6 MEMORY POOL MANAGEMENT FUNCTION

This chapter describes the memory pool management function of RX4000.

6.1 Overview

RX4000 statically generates and initializes those objects that are under its management during system initialization. These objects include information tables for managing the overall system and management blocks for implementing the functions (such as semaphores and event flags).

RX4000 is provided with a dynamic memory pool management function, so that memory areas can be acquired as required and released once they become unnecessary. The user can thus dynamically allocate the memory for objects, enabling the efficient use of the memory space.

6.2 Management Objects

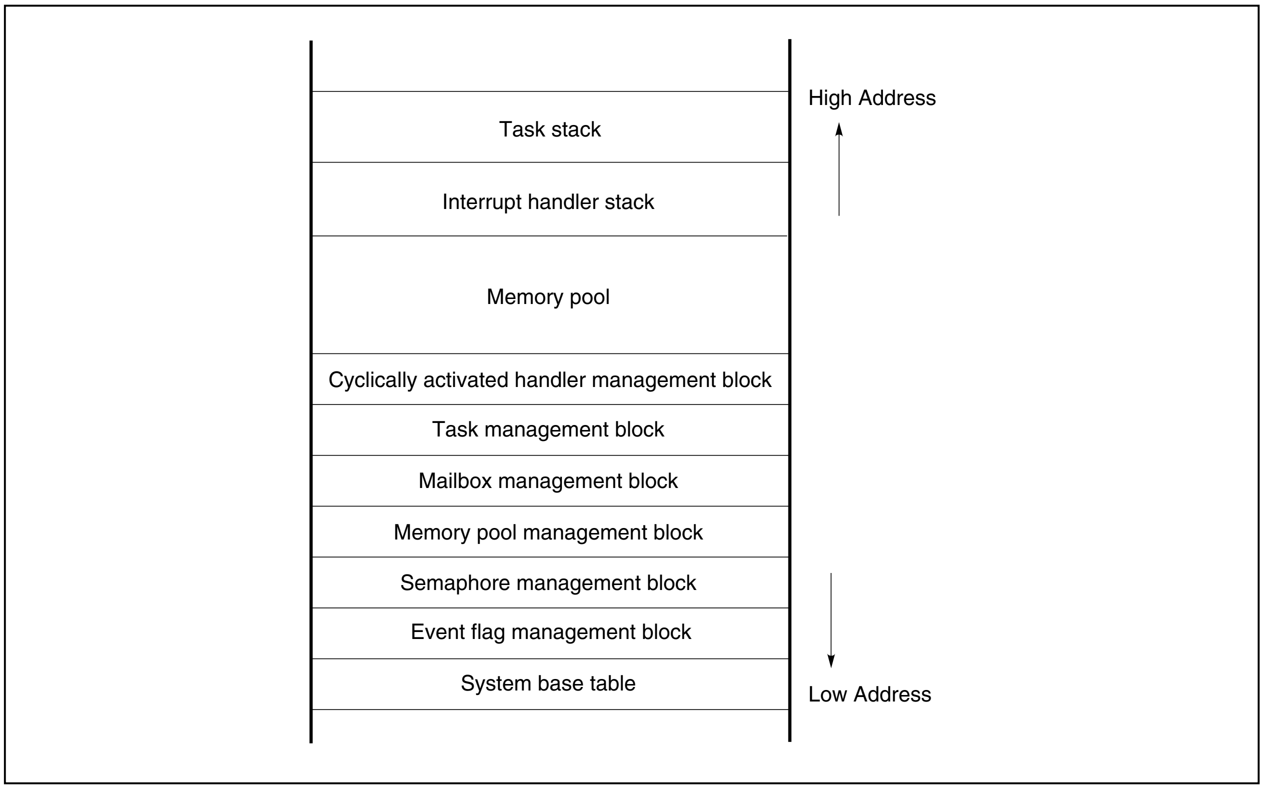
The following lists the objects required for implementing the functions provided by RX4000.

These management objects are generated and initialized during system initialization, according to the information specified at configuration (for tasks, semaphores, etc.).

- System base table
- Task management block
- Semaphore management block
- Event flag management block
- Mailbox management block
- Memory pool management block
- Memory block management block
- Cyclically activated handler management block
- Memory pool
- Task stack
- Interrupt handler stack

Figure 6-1 shows a typical arrangement of the management objects.

Figure 6-1. Typical Arrangement of Management Objects



6.3 Memory Pool and Memory Blocks

RX4000 performs dynamic memory pool management, so that a memory area can be provided for a processing program (task or interrupt handler) when requested. Furthermore, the memory pool offered by RX4000 has a variable length.

The memory pool consists of memory blocks and is allocated in units of memory blocks.

Dynamic generation of a memory pool and access to the memory pool are performed using the following memory pool-related system calls:

cre_mpl	: Generates a memory pool.
del_mpl	: Deletes the memory pool.
get_blk	: Acquires a memory block.
pget_blk	: Acquires a memory block (by polling).
tget_blk	: Acquires a memory block (with timeout setting).
rel_blk	: Release a memory block.
ref_mpl	: Acquires memory pool information.

6.3.1 Generating a memory pool

RX4000 provides two interfaces for generating (registering) a memory pool. One enables static generation during system initialization (in the nucleus initialization section). The other enables dynamic generation by issuing a system call from within a processing program.

To generate a memory pool with RX4000, certain areas in system memory are allocated to enable management of the memory pool (as an object of RX4000 management) and for the memory pool entity, then initialized.

(1) Static registration of a memory pool

To register a memory pool statically, specify it during configuration.

RX4000 generates the memory pool, based on the information defined in the information file (including system information tables and system information header subfiles) during system initialization. The memory pool is subsequently managed by RX4000.

(2) Dynamic registration of a memory pool

To dynamically register a memory pool, issue the cre_mpl system call from within a processing program (task).

RX4000 generates the memory pool, according to the information specified by the parameters when the cre_mpl system call is issued. The memory pool is subsequently managed by RX4000.

6.3.2 Deleting a memory pool

A memory pool is deleted upon the issue of a del_mpl system call.

- del_mpl system call

  The del_mpl system call deletes the memory pool specified by the parameter.

  Subsequently, that memory pool is no longer subject to management by RX4000.

  If a task is queued into the queue of the memory pool specified by this system call parameter, that task shall be removed from the queue, leave the wait state (the memory block wait state) then enter the ready state.

  E-DLT is returned, to the task that was released from the wait state, as the return value for the system call (get_blk or tget_blk) that triggered the transition of the task to the wait state.

6.3.3 Acquiring a memory block

A memory block is acquired by issuing a get_blk, pget_blk, or tget_blk system call.

Caution In RX4000, memory clear is not performed when a memory block is acquired. Therefore, the acquired memory block's contents are undefined.

- get_blk system call
  Upon the issue of the get_blk system call, the processing program (task) acquires a memory block from the memory pool specified by a parameter.
  After the issue of this system call, if the task cannot acquire the block from the specified memory pool (because no free block of the required size exists), the task itself is enqueued into the queue of this memory pool. Thus, the task leaves the run state and enters the wait state (the memory block wait state).
  The memory block wait state is canceled in the following cases and the task returns to the ready state.

  - When a rel_blk system call is issued and a memory block of the required size is returned.
  - When a del_mpl system call is issued and the specified memory pool is deleted.
  - When a rel_wai system call is issued and the wait state is forcibly canceled.
  - When a vsnd_sig system call is issued and the waits state is forcibly canceled.

Remark When a task queues in the wait queue of the specified memory pool, it is executed in the sequence (FIFO sequence or priority order sequence) specified when that memory pool was generated (during configuration or when a cre_mpl system call was issued).

- pget_blk system call
  Upon the issue of the pget_blk system call, the processing program (task) acquires a memory block from the memory pool specified by a parameter.
  For this system call, if the task cannot acquire the block from the memory pool specified by this system call parameter (because no free block of the required size exists), E_TMOUT is returned as the return value.

- tget_blk system call
  By issuing a tget_blk system call, the processing program (task) acquires a memory block from the memory pool specified by a parameter.
  After the issue of this system call, if the task cannot acquire the block from the specified memory pool (because no free block of the required size exists), the task itself is enqueued into the queue of this memory pool. Thus, the task leaves the run state and enters the wait state (the memory block wait state).
  The memory block wait state is canceled in the following cases and the task returns to the ready state.

  - When the wait time specified by a parameter has elapsed.
  - When a rel_blk system call is issued and a memory block of the required size is returned.
  - When a del_mpl system call is issued and the specified memory pool is deleted.
  - When a rel_wai system call is issued and the wait state is forcibly canceled.
  - When a vsnd_sig system call is issued and the waits state is forcibly canceled.

Remark When a task queues in the wait queue of the specified memory pool, it is executed in the sequence (FIFO sequence or priority order sequence) specified when that memory pool was generated (during configuration or when a cre_sem system call was issued).

6.3.4 Returning a memory block

A memory block is returned upon the issue of an rel_blk system call.

- rel_blk system call

  Upon the issue of an rel_blk system call, a processing program (task) returns a memory block to the memory pool specified by a parameter.

  For this system, if the memory block returned by this system call is of the size required by the first task in the queue of the specified memory pool, this block is passed to that task.

  Thus, the first task is removed from the queue, leaves the wait state (the memory block wait state), and enters the ready state. Or, it leaves the wait_suspend state and enters the suspend state.

Cautions 1. The contents of a returned memory block are not cleared automatically by RX4000. Thus, the contents of a memory block may be undefined when that memory block is returned.

2. Treat a memory pool which returns a memory block the same as a memory block specified when issuing the get_blk, pget_blk or tget_blk system call.

6.3.5 Acquiring memory pool information

Memory pool information is acquired by issuing an ref_mpl system call.

- ref_mpl system call

  Upon the issue of an ref_mpl system call, the processing program (task) acquires the memory pool information (extended information, queued tasks, etc.) for the memory pool specified by a parameter.

  The memory pool information consists of the following:

  - Extended information
  - Whether tasks are queued
  - Total amount of free space
  - The maximum memory block size to be acquired

[MEMO]

CHAPTER 7 TIME MANAGEMENT FUNCTION

This chapter describes the time management function of RX4000.

7.1 Overview

Time management of RX4000 is performed using CP0 timer interrupts which can be generated periodically by software.

If a timer interrupt is issued, the RX4000 system clock processing is called and system clock update as well as processing related to time, called timer operations (such as task delay rise, time out and starting of the period start handler) is executed.

7.2 System Clock

The system clock is a software timer that provides the time (in units of milliseconds, with a width of 48 bits) used for time management by RX4000.

The system clock is set to 0×0000 0000 0000 at system initialization and updated in units of the basic clock cycle (specified at configuration) each time system clock processing is performed.

Caution The system clock managed by RX4000 shall have a fixed width of 48 bits. RX4000 ignores any overflow (that exceeding 48 bits) for the clock value.

7.2.1 Setting and reading the system clock

The system clock setting is executed by issuing the set_tim system call, and reading by issuing the get_tim system call.

- set_tim system call
  The set_tim system call sets the time specified by a parameter to the system clock.

- get_tim system call
  The get_tim system call stores the current time of the system clock into the packet specified by a parameter.

7.3 Delayed Task Wake-Up

Delayed task wake-up changes the state of a task from run to wait (the timeout wait state) and leaves the task in this state for a given period. Once this period elapses, the task is released from the wait state and returns to the ready state.

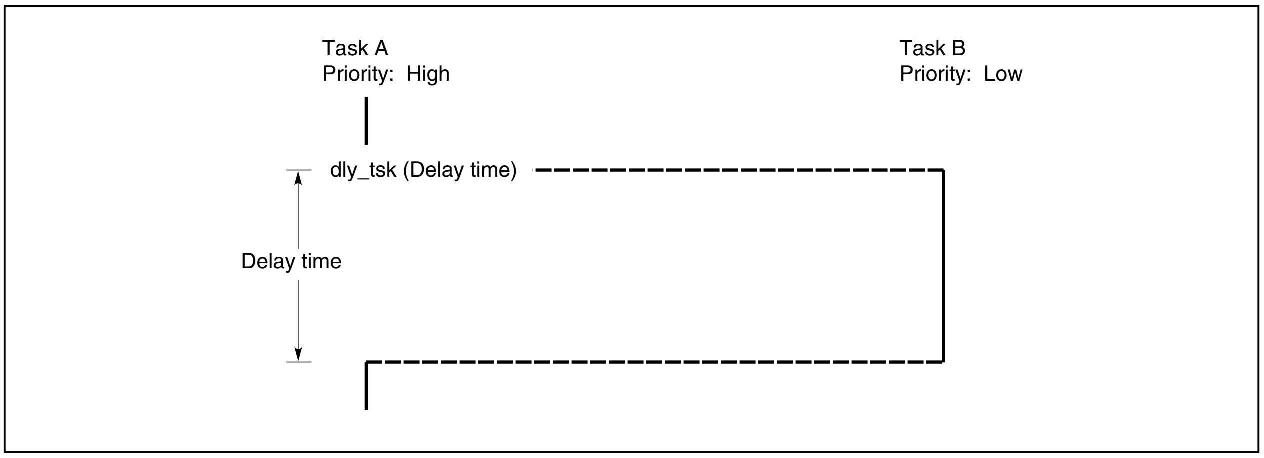
Delayed task wake-up is performed by issuing a dly_tsk system call.

- dly_tsk system call
  Upon the issue of a dly_tsk system call, the state of the task from which this system call was issued changes from run to wait (the timeout wait state).
  The timeout wait state is canceled in the following cases and the task returns to the ready state.

  - Upon the elapse of the delay specified by a parameter.
  - Upon the issue of a rel_wai system call and the forcible cancelation of the wait state.

Figure 7-1 shows the flow of the processing after the issue of the dly_tsk system call.

Figure 7-1. Flow of Processing After Issue of dly_tsk



7.4 Timeout

If the conditions required for a certain action are not satisfied when that action is requested by a task, the timeout function changes the state of the task from run to wait (wake-up wait state, resource wait state, etc.) and leaves the task in the wait state for a given period. Once that period elapses, the timeout function releases the task from the wait state. Then, the task returns to the ready state.

The timeout function is enabled by issuing a tslp_tsk, twai_sem, twai_flg, trcv_msg, or tget_blk system call.

- tslp_tsk system call

  Upon the issue of a tslp_tsk system call, one request for wake-up, issued for the task from which this system call is issued, is canceled (the wake-up request counter is decremented by 0x1).

  If the wake-up request counter of the task from which this system call is issued currently indicates 0x0, the wake-up request is not canceled (decrement of the wake-up request counter) and the task enters the wait state (the wake-up wait state) from the run state.

  The wake-up wait state is canceled in the following cases, and the task returns to the ready state.

  - When the given wait time specified by a parameter has elapsed.
  - When a wup_tsk system call is issued.
  - When a ret_wup system call is issued.
  - When a rel_wai system call is issued and the wait state is forcibly canceled.
  - When a vsnd_sig system call is issued and the wait state is forcibly canceled.

- twai_sem system call

  Upon the issue of a twai_sem system call, the task acquires a resource from the semaphore specified by a parameter (the semaphore counter is decremented by 0x1).

  After the issue of this system call, if the task cannot acquire a resource from the semaphore specified by a parameter (no free resource exists), the task itself is enqueued into the queue of this semaphore. Thus, the task leaves the run state and enters the wait state (the resource wait state).

  The resource wait state is canceled in the following cases, and the task returns to the ready state.

  - When the given wait time specified by a parameter has elapsed.
  - When a sig_sem system call is issued.
  - When a del_sem system call is issued and the specified semaphore is deleted.
  - When a rel_wai system call is issued and the wait state is forcibly canceled.
  - When a vsnd_sig system call is issued and the wait state is forcibly canceled.

- twai_flg system call

  The twai_flg system call checks whether the bit pattern is set so as to satisfy the wait condition required for the event flag specified by a parameter.

  If the bit pattern does not satisfy the wait condition required for the event flag specified by this system call parameter, the task from which this system call is issued is enqueued at the end of the queue of this event flag. Thus, the task leaves the run state and enters the wait state (the event flag wait state).

  The event flag wait state is canceled in the following cases, and the task returns to the ready state.

  - When the given wait time specified by a parameter has elapsed.
  - When a set_flg system call is issued and the required wait condition is satisfied.
  - When a del_flg system call is issued and the specified event flag is deleted.
  - When a rel_wai system call is issued and the wait state is forcibly canceled.
  - When a vsnd_sig system call is issued and the wait state is forcibly canceled.

- trcv_msg system call

  Upon the issue of a trcv_msg system call, the task receives a message from the mailbox specified by a parameter.

  After the issue of this system call, if the task cannot receive a message from the specified mailbox (no messages exist in the message queue of that mailbox), the task itself is enqueued at the end of the task queue of this mailbox. Thus, the task leaves the run state and enters the wait state (the message wait state).

  The message wait state is canceled in the following cases, and the task returns to the ready state.

  - When the given time specified by a parameter has elapsed.
  - When a snd_msg system call is issued.
  - When a del_mbx system call is issued and this mailbox is deleted.
  - When a rel_wai system call is issued and the wait state is forcibly canceled.
  - When a vsnd_sig system call is issued and the wait state is forcibly canceled.

- tget_blk system call

  Upon the issue of a tget_blk system call, the task acquires a memory block from the memory pool specified by a parameter.

  After the issue of this system call, if the task cannot acquire the block from the specified memory pool (because no free block of the required size exists), the task itself is enqueued into the queue of this memory pool. Thus, the task leaves the run state and enters the wait state (the memory block wait state).

  The memory block wait state is canceled in the following cases, and the task returns to the ready state.

  - When the given wait time specified by a parameter has elapsed.
  - When a rel_blk system call is issued and a memory block of the required size is returned.
  - When a del_mpl system call is issued and the specified memory pool is deleted.
  - When a rel_wai system call is issued and the wait state is forcibly canceled.
  - When a vsnd_sig system call is issued and the wait state is forcibly canceled.

7.5 Cyclically Activated Handler

The cyclically activated handler is an exclusive period processing routine which starts immediately when a predetermined start time arrives, and is a processing program which has optimally small overhead within the periodic processing program described by the user until execution is started.

The cyclically activated handler is treated as independent of the task. For this reason, even if a task with the highest priority order is being executed in the system, that processing is interrupted and the system switches to the cyclically activated handler's control.

The following system calls and instructions relevant to a cyclically activated handler are used in the dynamic operation of a cyclically activated handler.

def_cyc	: Registers a cyclically activated handler.
act_cyc	: Controls the activity state of the cyclically activated handler.
ref_cyc	: Acquires cyclically activated handler information.
return	: Performs return from the cyclically activated handler.

7.5.1 Registering a cyclically activated handler

RX4000 provides two interfaces for registering a cyclically activated handler. One enables static registration during system initialization (in the nucleus initialization section). The other enables dynamic registration by issuing a system call from within a processing program.

To register a cyclically activated handler with RX4000, an area in system memory shall be allocated for managing the cyclically activated handler (to be managed by RX4000), then initialized.

(1) Static registration of a cyclically activated handler

To statically register a cyclically activated handler, specify it during configuration.

RX4000 performs the processing for registering the cyclically activated handler, based on the information defined in the information file (including system information tables and system information header subfiles) during system initialization. The cyclically activated handler is subsequently managed by RX4000.

(2) Dynamic registration of a cyclically activated handler

To dynamically register a cyclically activated handler, issue the def_cyc system call from within a processing program (task or non-task).

RX4000 performs the processing for registering the cyclically activated handler, according to the information specified by a parameter when the def_cyc system call is issued.

The cyclically activated handler is subsequently managed by RX4000.

7.5.2 Activity state of cyclically activated handler

The activity state of a cyclically activated handler is used as a criterion for determining whether RX4000 initiates the cyclically activated handler.

The activity state is set when the cyclically activated handler is registered (during configuration or when a def_cyc system call is issued). However, RX4000 allows the user to change the activity state of the cyclically activated handler from a user processing program.

- act_cyc system call

  Upon the issue of an act_cyc system call, the activity state of the cyclically activated handler is switched ON/OFF, as specified with the parameter.

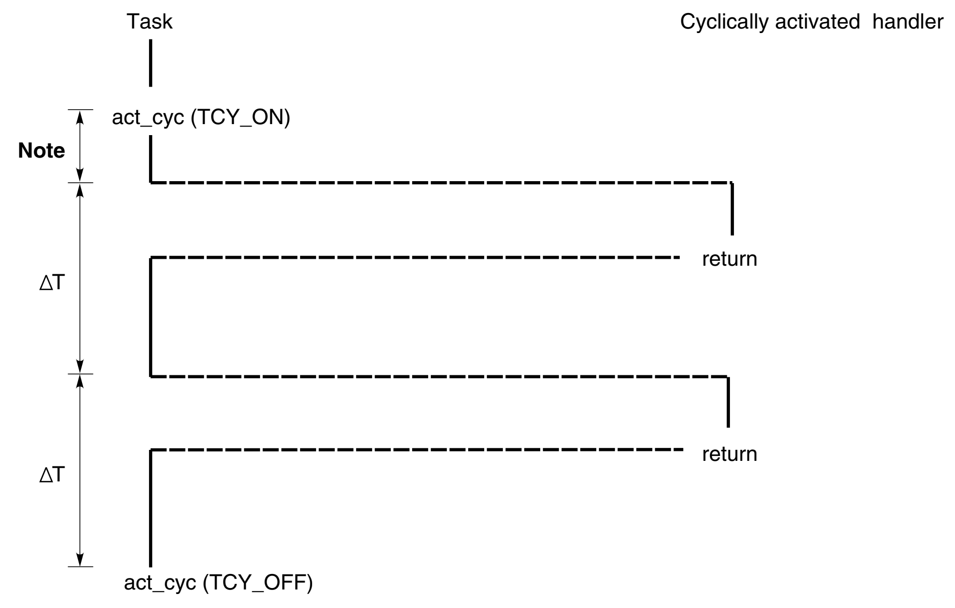
TCY_OFF	:	Switches the activity state of the cyclically activated handler to OFF.
TCY_ON	:	Switches the activity state of the cyclically activated handler to ON.
TCY_INI	:	Initializes the cycle counter of the cyclically activated handler.

While RX4000 is running, the cycle counter continues to count even when the activity state of the cyclically activated handler is OFF. In some cases, when an act_cyc system call is issued to switch the activity state of the cyclically activated handler from OFF to ON, the first restart request could be issued sooner than the activation time interval specified when it was registered (during configuration or upon the issue of the def_cyc system call). To prevent this, the user must specify (TCY_INI) to initialize the cycle counter as well as (TCY_ON) to restart the cyclically activated handler when issuing the act_cyc system call. Then, the first restart request will be issued in sync with the time interval, specified when it was registered.

Figures 7-2 and 7-3 show the flow of the processing after the issue of an act_cyc system call from a processing program to switch the activity state of the cyclically activated handler from OFF to ON.

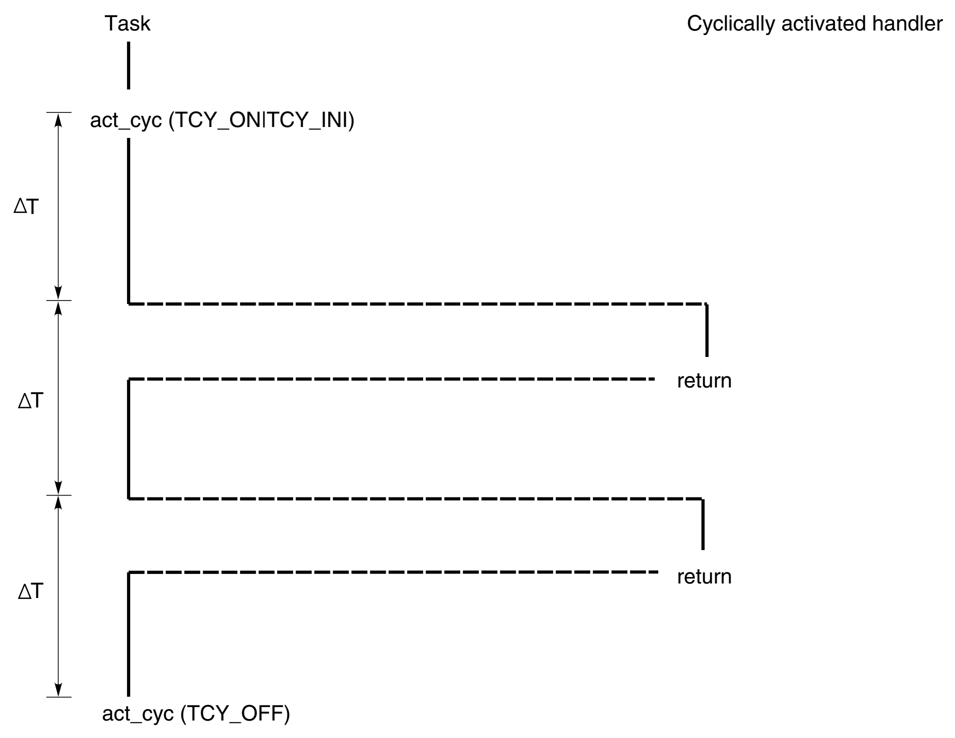
In the figures, ΔT is assumed to be the activation time interval, specified for the cyclically activated handler when it was registered.

Figure 7-2. Flow of Processing After Issue of act_cyc (TCY_ON)



Note ΔT is the time until counting by the cycle counter is finished.

Figure 7-3. Flow of Processing After Issue of act_cyc (TCY_ON|TCY_INI)



7.5.3 Internal processing performed by cyclically activated handler

After the occurrence of a timer interrupt, RX4000 performs preprocessing for interruption before control is passed to the cyclically activated handler. When control is returned from the cyclically activated handler, RX4000 performs interrupt post processing.

When describing the processing to be performed by the activated interrupt handler, note the following:

(1) Saving/restoring the registers

Based on the function call protocol for VR Series cross tool, RX4000 saves the work registers when control is passed to the cyclically activated handler, and restores them upon the return of control from the handler. Therefore, the cyclically activated handler does not have to save the work registers when it starts, nor restore them upon the completion of its processing. Save/restoration of the registers should not be coded in the description of the cyclically activated handler.

(2) Stack switching

RX4000 performs stack switching when control is passed to the cyclically activated handler and upon a return from the handler. Therefore, the cyclically activated handler does not have to switch to the interrupt handler stack when it starts, nor switch to the original stack upon the completion of its processing. However, if the interrupt handler stack is not defined during configuration, stack switching is not performed and system continues to use that stack being used upon the occurrence of an interrupt.

(3) Limitations imposed on system calls

There are some RX4000 system calls which cannot be executed within the cyclically activated handler.

The following lists the system calls that can be issued during the processing performed by a cyclically activated handler:

- Task management system calls

sta_tsk	chg_pri	rot_rdq	rel_wai	get_tid
rer_tsk				

- Task-associated handler function system call

vsnd_sig				

- Task-associated synchronization system calls

sus_tsk	rsm_tsk	frsm_tsk	wup_tsk	can_wup

- Synchronous communication system calls

sig_sem	preq_sem	ref_sem	set_flg	clr_flg
pol_flg	ref_flg	snd_msg	prcv_msg	ref_mbx

- Interrupt management system calls

def_int	ret_int	ret_wup	dis_int	ena_int
chg_ims	ref_ims			

- Memory pool management system calls

pget_blk	rel_blk	ref_mpl		

- System management system calls

get_ver	ref_sys	def_svc	viss_svc	def_exc

(4) Return processing from the cyclically activated handler

Return processing from the cyclically activated handler is performed by issuing an return instruction upon the completion of the processing performed by cyclically activated handler.

When a system call (chg_pri, sig_sem, etc.) that requires task scheduling is issued during the processing of a cyclically activated handler, RX4000 merely queues that task into the queue. The actual task scheduling is batched and deferred until return from the cyclically activated handler has been completed (by issuing an return instruction).

7.5.4 Acquiring cyclically activated handler information

Information related to a cyclically activated handler is acquired by issuing an ref_cyc system call.

- ref_cyc system call

  By issuing an ref_cyc system call, the task acquires information (including extended information, remaining time, etc.) related to the cyclically activated handler specified by a parameter.

  The cyclically activated handler information consists of the following:

  - Extended information
  - Time remaining until the next start of the cyclically activated handler
  - Current activity state

CHAPTER 8 SCHEDULER

This chapter explains the task scheduling performed by RX4000.

8.1 Overview

By monitoring the dynamically changing task states, the RX4000 scheduler manages and determines the sequence in which tasks are executed, and assigns a processing time to a specific task.

8.2 Drive Method

The RX4000 scheduler uses an event-driven technique, in which the scheduler operates in response to the occurrence of some event.

The “occurrence of some event” means the issue of a system call that may cause a task state change, the issue of a return instruction that causes a return from a handler, or the occurrence of a clock interrupt.

When these phenomena occur, task scheduling processing is executed with the scheduler driving.

The following system calls can be used to drive the scheduler.

- Task management system calls

  sta_tsk   ext_tsk   exd_tsk   ena_dsp   chg_pri

  rot_rdq   rel_wai

- Task-associated synchronization system calls

  rsm_tsk   frsm_tsk   slp_tsk   tslp_tsk   wup_tsk

- Synchronous communication system calls

  del_sem   sig_sem   wai_sem   twai_sem   del_flg

  set_flg   wai_flg   twai_flg   del_mbx   snd_msg

  rcv_msg   trcv_msg

- Interrupt management system calls

  ret_int   ret_wup   unl_cpu

- Memory pool management system calls

  del_mpl   get_blk   tget_blk   rel_blk

- Time management system calls

  dly_tsk

8.3 Scheduling Method

RX4000 uses the priority and FCFS (First-Come, First-Served) scheduling method. When driven, the scheduler checks the priority of each task that can be executed (in the ready state), selects the optimum task, and assigns a processing time to the selected task.

8.3.1 Priority method

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

The scheduler checks the priority of each task that can be executed (in the ready state), selects the task having the highest priority, and assigns a processing time to the selected task.

Remark In RX4000, a task to which a smaller value is assigned as the priority level has a higher priority.

8.3.2 FCFS method

RX4000 can assign the same priority to more than one task. Because the priority method is used for task scheduling, there is the possibility of more than one task having the highest priority being selected.

Among those tasks having the highest priority, the scheduler selects the first to become executable (that task which has been in the ready state for the longest time) and assigns a processing time to the selected task.

8.4 Idle Handler

The idle handler is started from the scheduler if all the tasks are not in the run state or not in the ready state, that is, if there is not even one task which is an object of RX4000 scheduling.

The idle handler is a processing routine prepared for utilize the power mode functions offered by the VR4100 effectively.

(1) Idle handler generation and activation

The idle handler is generated by system initialization (nucleus initialization section) and is started from the scheduler.

The idle handler is a handler defined by RX4000 and operations (generation, activating, terminating, deleting, etc.) cannot be executed with respect to the idle handler from a user's processing program (task/non-task).

(2) Processing within the idle handler

The role of the idle handler is to switch the processor to the low power mode. Then the idle handler issues a STANDBY, SUSPEND, or HIBERNATE command.

Furthermore, the processor low power mode is canceled for the following two reasons.

- Issue of an external interrupt (maskable interrupt, nonmaskable interrupt).
  If an interrupt is issued, the relevant interrupt handler is started and the processor is returned from the low power mode to the normal mode.
  However, in an interrupt handler which corresponds to a nonmaskable interrupt, issue of a system call is prohibited, so when processing is completed, the processor is again switched to the low power mode.

- Reset
  In the reset sequence, since operation starts from initialization processing (boot processing), the low power mode is canceled.

8.5 Implementing a Round-Robin Method

In scheduling based on the priority and FCFS methods, even if tasks have the same priority as that currently running, they cannot be executed unless that task to which a processing time has been assigned first enters another state or relinquishes control of the processor.

RX4000 provides system calls such as rot_rdq to implement a scheduling method (round-robin method) that can overcome the problem incurred by the priority and FCFS methods.

The round-robin method can be implemented as follows:

Conditions

- Task priority
  Task A = task B = task C

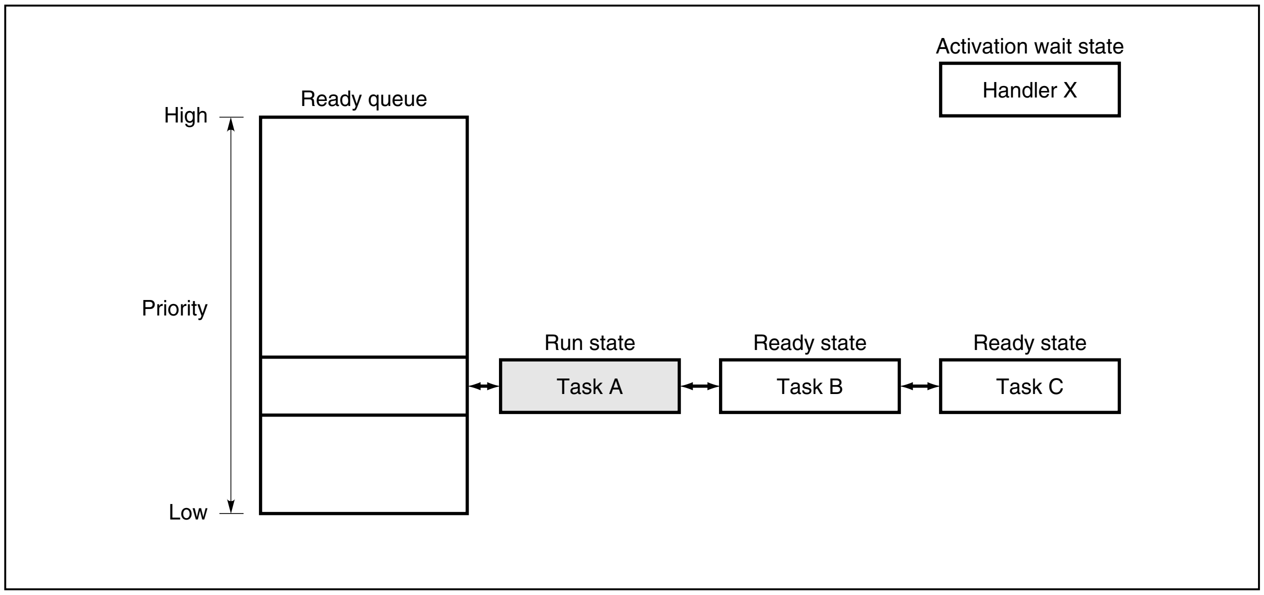
- State of tasks
  Task A: run state
  Task B: ready state
  Task C: ready state

- Cyclically activated handler X attributes

Activity state:	ON
Activation interval:	ΔT (unit: basic clock period)
Processing:	Rotation of the ready queues (issue of the rot_rdq system call)

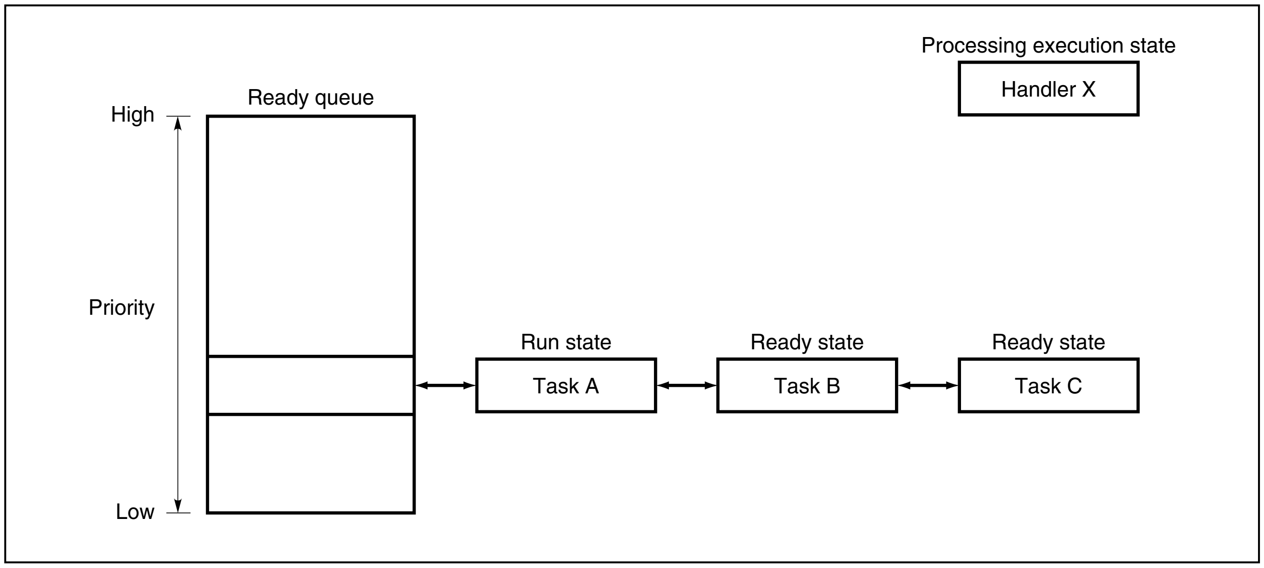
(1) Task A is currently running.
The other tasks (B and C) have the same priority as task A, but they cannot be executed unless task A enters another state or relinquishes control of the processor.
The ready queue becomes as shown in Figure 8-1.

Figure 8-1. Ready Queue State (1)



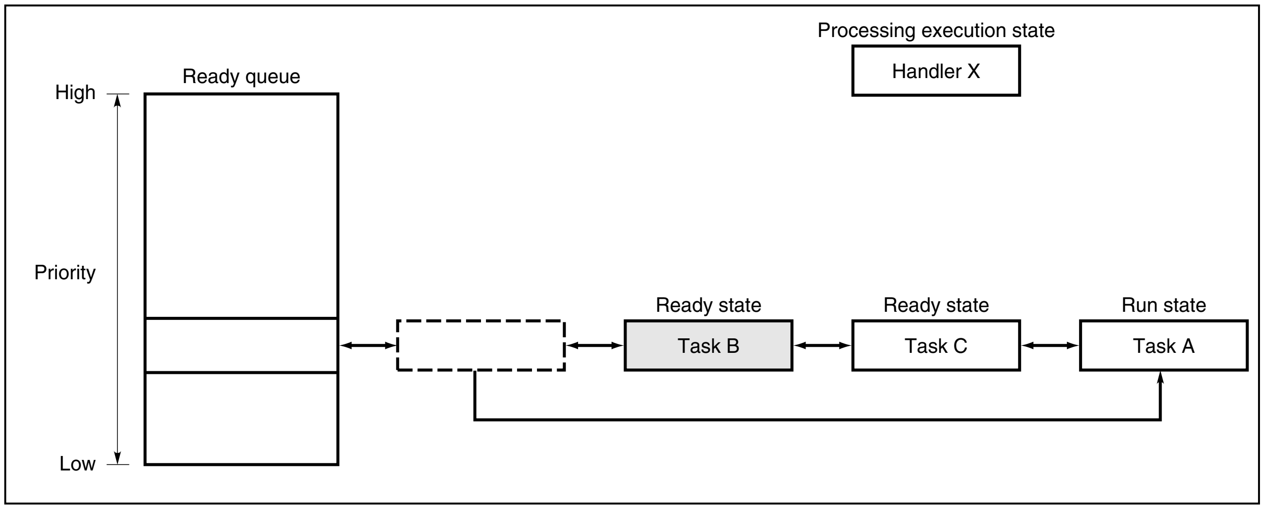
(2) Cyclically activated handler X starts when the predetermined period of time has passed.
In this way, processing of task A is interrupted and cyclically activated handler processing is executed.
The ready queue changes to the state shown in Figure 8-2.

Figure 8-2. Ready Queue State (2)



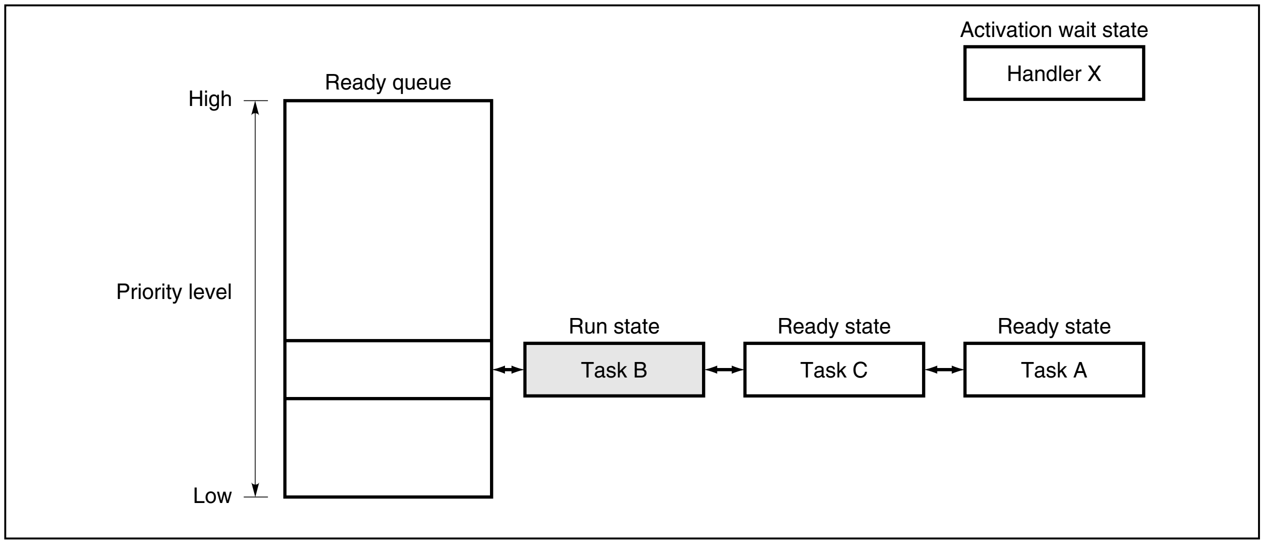
(3) Cyclically activated handler X issues a rot_rdq system call.
In this way, task A is queued at the tail end of the ready queue in accordance with its priority level.
The ready queue changes to the state shown in Figure 8-3.

Figure 8-3. Ready Queue State (3)



(4) Cyclically activated handler X issues a return instruction and processing is terminated.
In this way, task A changes from the run state to the ready state and task B changes from the ready state to the run state.
Figure 8-4 shows the ready queue state at this time.

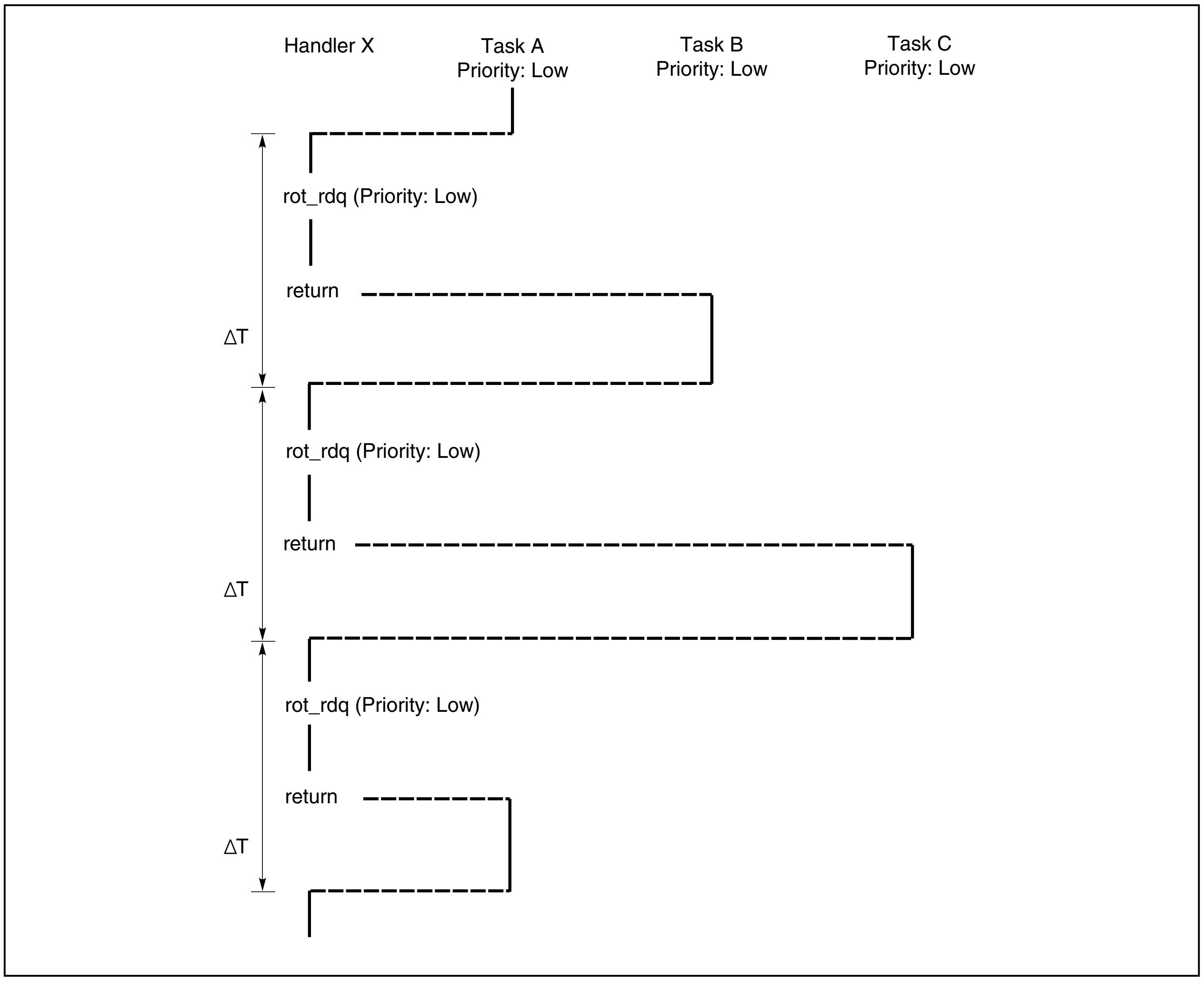
Figure 8-4. Ready Queue State (4)



(5) By issuing the rot_rdq system call from the cyclically activated handler that is started at constant intervals, that scheduling method (round-robin method) in which tasks are switched every time the specified period (ΔT) elapses is implemented.

Figure 8-5 shows the processing flow when the round-robin method is used.

Figure 8-5. Scheduling When the Round-Robin Method Is Used



8.6 Scheduling Lock Function

In RX4000 a function is offered which drives the scheduler from a user processing program (task) and which disables or enables task scheduling processing (dispatch processing).

These functions are implemented by issuing the following system calls from within a task.

- dis_dsp system call
  Disables dispatching (task scheduling).
  If this system call is issued, control is not passed to another task until the ena_dsp system call is issued.

- ena_dsp system call
  Enable dispatching (task scheduling).
  When this system call is issued, dispatching which was disabled by the issue of the dis_dsp system call is resumed.
  When the dis_dsp system call has been issued, if a system call that requires task scheduling (such as chg_pri or sig_sem) is issued, RX4000 merely executes processing such as queue operation until the ena_dsp system call is issued. Actual scheduling is delayed and executed at one time upon the issue of the ena_dsp system call.

- loc_cpu system call
  Disables the acceptance of maskable interrupts, then disables dispatching (task scheduling).
  If this system call is issued, control will not be passed to another task or handler until the unl_cpu system call is issued.

- unl_cpu system call
  Enables the acceptance of maskable interrupts, then restarts dispatching (task scheduling).
  If a maskable interrupt has occurred between the issue of the loc_cpu system call and that of the unl_cpu system call, transfer of control to the corresponding interrupt handling (processing of the interrupt handler) is delayed until unl_cpu system call is issued. Also, if a system call which is necessary for task scheduling processing (such as chg_pri or sig_sem) is issued during the interval after the loc_cpu system call is issued and until the unl_cpu system call is issued, only processing of wait queue operations is delayed until the unl_cpu system call is issued, being performed by batch processing.

The flow of control if scheduling processing is not delayed is shown in Figure 8-6 and the flow of control if the dis_dsp and loc_cpu system calls are issued is shown in Figure 8-7 and Figure 8-8.

Figure 8-6. Flow of Control if Scheduling Processing is Not Delayed

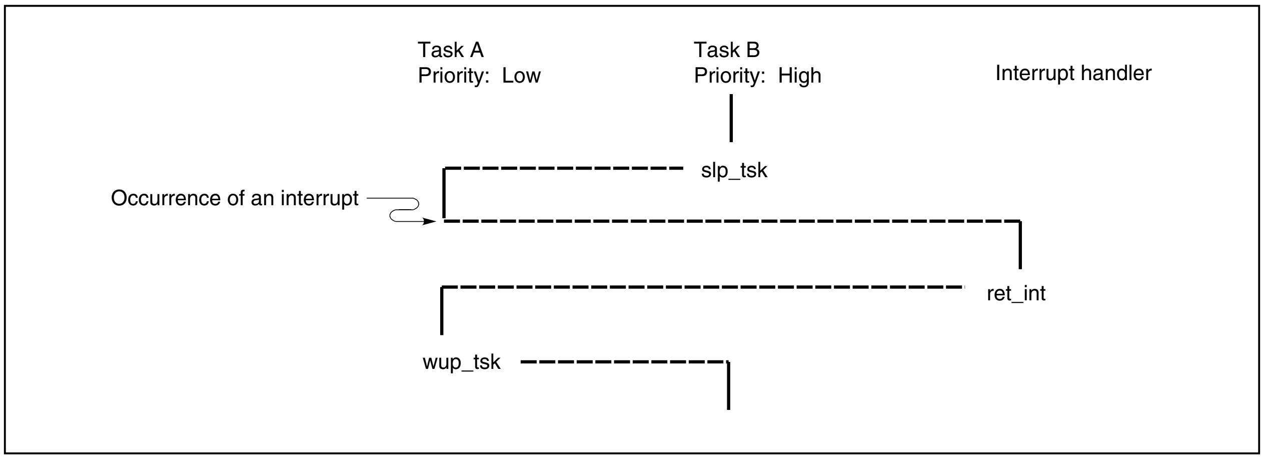


Figure 8-7. Flow of Control if the dis_dsp System Call is Issued

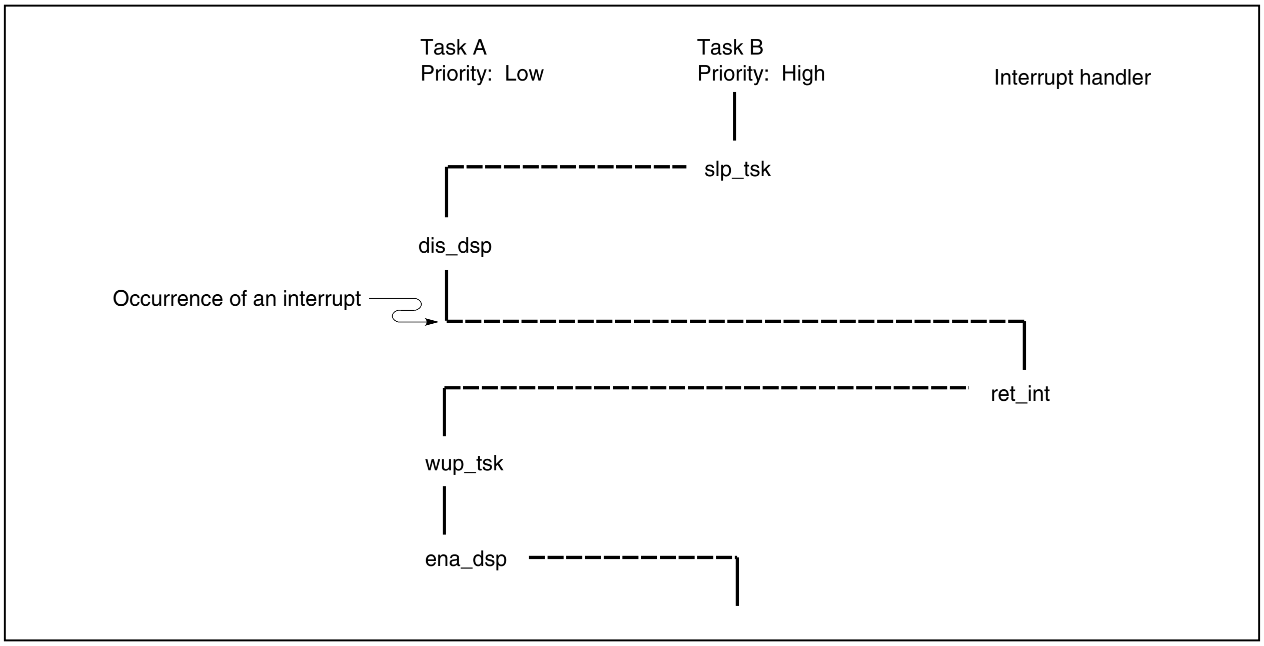
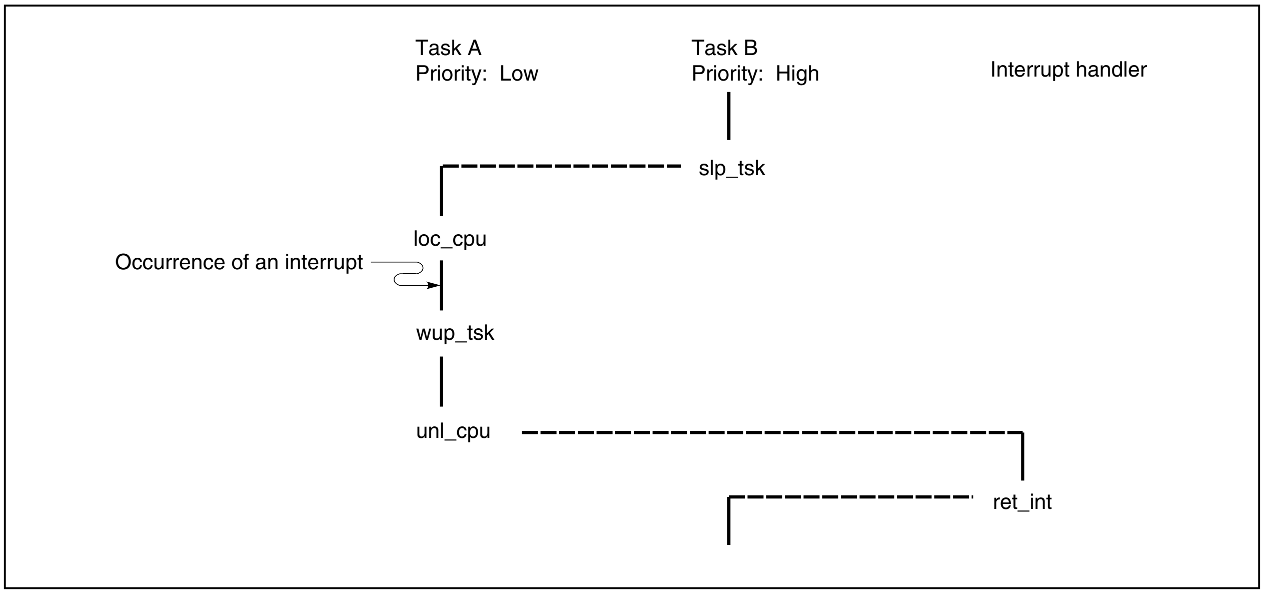


Figure 8-8. Flow of Control if the loc_cpu System Call is Issued



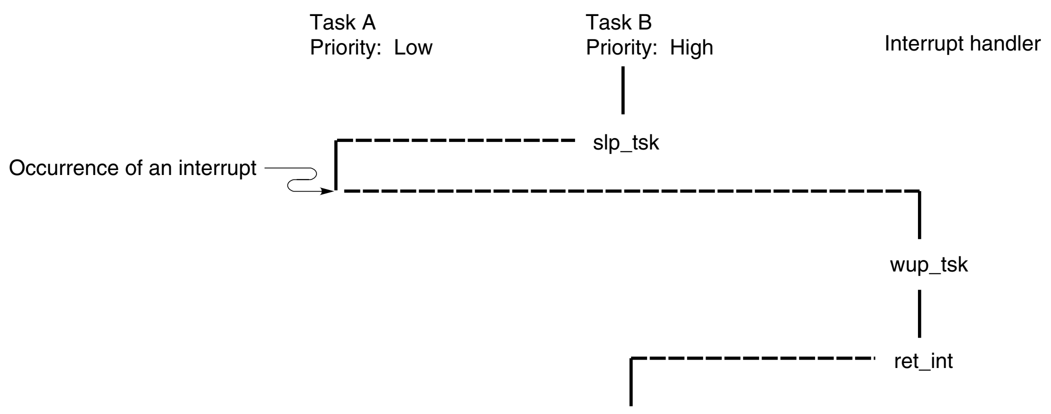
8.7 Scheduling While the Handler Is Operating

To quickly terminate handlers (interrupt handlers and cyclically activated handlers), RX4000 delays the driving of the scheduler until processing within the handler terminates.

Therefore, if a system call that requires task scheduling (such as chg_pri or sig_sem) is issued, RX4000 merely executes processing such as queue operation until the completion of return processing from the handler (such as ret_int system call or the issue of return instruction). Actual scheduling is delayed and executed at one time upon the completion of return processing.

Figure 8-9 shows the control flow when a handler issues a system call that requires scheduling.

Figure 8-9. Flow of Control if the wup_tsk System Call is Issued



[MEMO]

CHAPTER 9 SYSTEM MANAGEMENT FUNCTIONS

This chapter describes system management functions performed by RX4000.

9.1 Overview

The following processing is performed by RX4000 in system management.

- System information management
- Extended SVC handler management
- Exception handler management

9.2 System Information Management

Acquires system information about the RX4000 version or system status (task execution conditions, interrupt enabled/disabled state, etc.) through the issuing of system calls.

9.3 Extended SVC Handler Management

Registers and issues extended SVC handlers, etc.

An extended SVC handler is a function which is defined as an extended system call by the user. However, an exclusive interface library is necessary for calling an extended SVC handler.

For details, refer to the **RX4000 User's Manual Installation**.

9.4 Exception Handler Management

The exception handler is an exclusive exception processing routine which is started immediately when an exception occurs, and is positioned as an extension of the processing program which issued the exception (task/non-task).

In RX4000, two types of exception handler interface, one for the CPU exception handler and the other for the system call exception handler, are offered. However, these handlers can only be registered one at a time for each application system.

(1) CPU exception handler

The CPU exception handler is a processing routine which is started when a CPU exception occurs.

When a CPU exception occurs, RX4000 transfers CPU exception information with the following structure to the exception handler as arguments.

- Structure of CPU exception information T_EXCCPUINFO

```
typedef struct t_exccpuinfo {
  ID      tskid;          /* Task ID No. */
  VW      cause;          /* Cause of exception */
  VP      pc;             /* Virtual address of instruction which caused the exception */
} T_EXCCPUINFO;
```

However, if a CPU exception occurs in a non-task, 0x0 is transferred as the task ID No.

(2) System call exception handler

The system call exception handler is a processing routine which starts when an exception occurs due to the issue of a system call.

In RX4000, if a system call exception occurs, system call exception information with the following structure is transferred to the exception handler as arguments.

- Structure of system call exception information T_EXCSYSINFO

```
typedef struct t_excsysinfo {
  ID      tskid;          /* Task ID No. */
  FN      sysno;          /* Function code */
  ER      ercd;           /* Error code */
  VP      pc;             /* Virtual address of system call which caused the exception */
} T_EXCSYSINFO;
```

However, if a system call exception occurs in a non-task, 0x0 is transferred as the task ID No.

9.4.1 Exception handler registration

In RX4000, two types of interface are provided for registering exception handlers, one for “static registration in system initialization (nucleus initialization section)” and the other for “issuing system calls from inside the processing program and dynamic registration.”

In RX4000, exception handler registration is securing an area in system memory for managing the exception handler (management object), then initializing it.

(1) In the case of static registration

In the case of static registration of an exception handler, it is specified when the system is being configured.

In RX4000, during system initialization, exception handler registration processing is performed based on the information defined in the information file (system information table, system information header file) and becomes a management object.

(2) In the case of dynamic registration

In the case of dynamic registration, the def_exc system call is issued from inside the processing program (task/non-task).

In RX4000, when the def_exc system call is issued, exception handler registration processing is performed based on the information specified in the parameters, and becomes a management object.

CHAPTER 10 SYSTEM INITIALIZATION

This chapter explains the system initialization performed by RX4000.

For details of the system initialization, refer to the **RX4000 User's Manual Installation**.

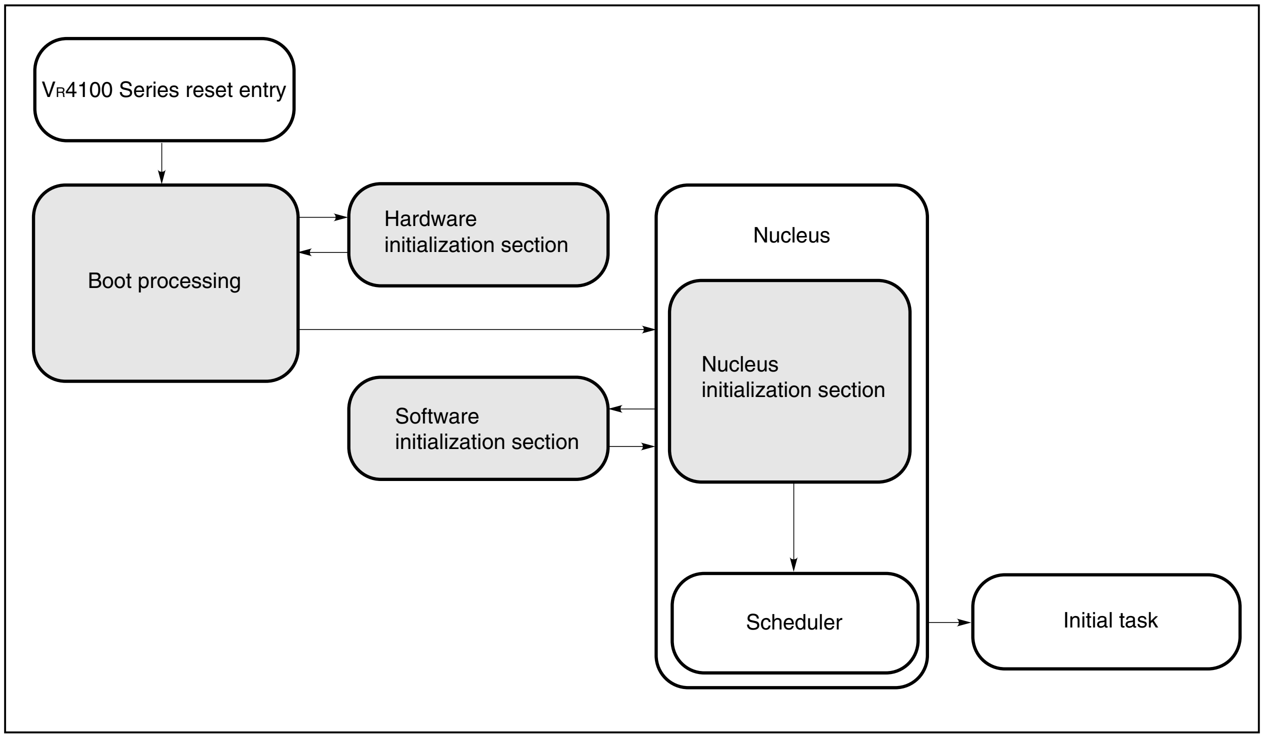
10.1 Overview

System initialization consists of initializing the hardware required by RX4000, as well as initializing the software.

Namely, in RX4000, the processing performed immediately after the system has been started is system initialization.

Figure 10-1 shows the flow of system initialization.

Figure 10-1. Flow of System Initialization


VR4100 Series reset entry
Boot processing
Hardware initialization section
Software initialization section
Nucleus
Nucleus initialization section
Scheduler
Initial task

10.2 Boot Processing

Boot processing is the first function executed in system initialization.

Boot processing involves the following:

- Setting of the gp registers
- Initialization of a memory area without initial values
- Calling of the hardware initialization section
- Transfer of control to the nucleus initialization section

10.3 Hardware Initialization Section

The hardware initialization section is a function for initializing the hardware in the execution environment (target system), and the section performs copying of initialization data. Since other contents are dependent on hardware configuration of execution environment, the user should describe the hardware initialization section.

10.4 Nucleus Initialization Section

The nucleus initialization section generates and initializes the management objects based on the information (such as task information or semaphore information) described in the information files (system information table and system information header file).

The nucleus initialization section performs the following processing:

- Interrupt initialization
- Timer initialization
- Memory pool generation and initialization
- Generation/initialization of management objects
- Activation of an initial task
- Calling of the software initialization section
- Transfer of control to the scheduler

10.5 Software Initialization Section

The software initialization section is a function for arranging the user’s software environment.

The software initialization section performs the following processing:

- Enabling of a clock interrupt
- Returns control to the nucleus initialization section

CHAPTER 11 INTERFACE LIBRARY

This chapter explains the interface library.

For details of the interface library, refer to the **RX4000 User's Manual Installation**.

11.1 Overview

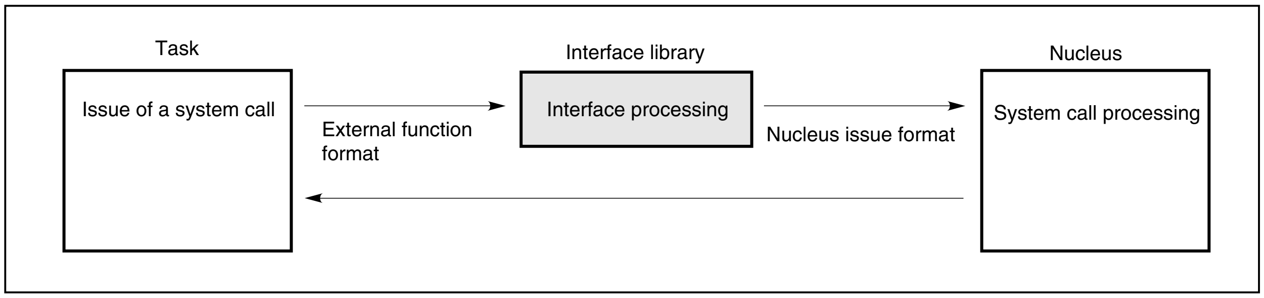
In RX4000, an interface library is provided which is positioned midway between the user processing program and the RX4000 nucleus. The interface library has a function for transferring control after performing setting of each type of necessary information, etc. for enabling processing by the nucleus.

When a processing program (task/non-task) is written in C, external function format is used to issue a system call or to call an extended SVC handler. The issue format that the nucleus can understand (nucleus issue format), however, differs from the external function format.

Then it becomes necessary to carry out the procedure for converting the system call issue format or expanded SVC handler calling format from the external function format to the nucleus issuing format (interfacing). There is an interface which performs the role of intermediary between the processing program and the nucleus for each system call. All these interfaces collected together are called the interface library.

Figure 11-1 shows the positioning of the interface library in RX4000.

Figure 11-1. Positioning of Interface Library



By providing an interface library, it becomes easy to separate the nucleus and the user processing program. For example, even if it becomes necessary to change the user's processing program after the nucleus body has been loaded in ROM, it becomes unnecessary to change the ROM where the nucleus body is stored. It also becomes possible to create it with the load module divided.

In RX4000, an interface library is provided which is compatible with the following cross tools for the V_R Series.

- For CodeWarrior (Metroworks Corporation)
- For C Cross MIPE Compiler (Green Hills Software, Inc.)

11.2 Processing in the Interface Library

The following processing is performed in the interface library.

- Setting of the necessary information in tables managed by the nucleus.
- Setting the necessary data in registers.
- After setting system call error values (However, errors set in the nucleus are excepted), it returns to the processing program.

11.3 Types of Interface Library

There are two types of interface library offered with RX4000, one with a function for checking system call parameters, and one without this function. The type of interface library which will be incorporated is specified during configuration.

The use of library with the parameter check function always return return values, if the parameters specified when a system call is issued are incorrect. On the other hand, the use of library without the parameter check function may not return return values, if the parameters specified when a system call is issued are incorrect.

Utilization of these two library types can be divided in accordance with the use. For example, during debugging, by use of the library with the parameter check function and by use of the library without the parameter check function during the actual build-in, improvements in program performance and capacity reductions can be realized.

Remark Errors in which return values are returned with the library which does not have a parameter check function are marked by “*” in the system call return value column in **CHAPTER 12 SYSTEM CALLS**.

Caution When the library without the parameter check function is used, if errors occur in which return values are not returned, the operation of the application system cannot be guaranteed.

CHAPTER 12 SYSTEM CALLS

This chapter describes the system calls supported by RX4000.

12.1 Overview

A system call is a procedure or function for invoking RX4000 service routines from the user’s processing programs (tasks/non-tasks). The user can use system calls to indirectly manipulate those resources (such as counters and queues) that are managed directly by RX4000.

RX4000 supports its own six system calls as well as the 68 defined in the μITRON3.0 specifications, thus enhancing the versatility of application systems.

System calls can be classified into the following eight groups, according to their functions.

(1) Task management system calls (13)

cre_tsk	del_tsk	sta_tsk	ext_tsk	exd_tsk
ter_tsk	dis_dsp	ena_dsp	chg_pri	rot_rdq
rel_wai	get_tid	ref_tsk		

These system calls are used to manipulate the status of a task.

This group provides functions for creating, activating, terminating, and deleting a task, a function for enabling and disabling dispatch processing, a function for changing the task priority, a function for rotating a task ready queue, a function for forcibly releasing a task from the wait state, and a function for referencing the task status.

(2) Task-associated handler function system calls (5)

vdef_sig	vret_sig	vsnd_sig	vchg_sms	vref_sms

This is the group of system calls which perform handler operations associated to tasks.

(3) Task-associated synchronization system calls (7)

sus_tsk	rsm_tsk	frsm_tsk	slp_tsk	tslp_tsk
wup_tsk	can_wup			

These system calls perform synchronous operations associated with tasks.

This group provides a function for placing a task in the suspend state and restarting a suspend task, a function for placing a task in the wake-up wait state and waking up a task currently in the wake-up wait state, and another function for canceling a task wake-up request.

(4) Synchronous communication system calls (22)

cre_sem	del_sem	sig_sem	wai_sem	preq_sem
twai_sem	ref_sem	cre_flg	del_flg	set_flg
clr_flg	wai_flg	pol_flg	twai_flg	ref_flg
cre_mbx	del_mbx	snd_msg	rcv_msg	prcv_msg
trcv_msg	ref_mbx			

These system calls are used for the synchronization (exclusive control and queuing) and communication between tasks.
This group provides a function for manipulating semaphores, a function for manipulating events and flags, and a function for manipulating mailboxes.

(5) Interrupt management system calls (9)

def_int	ret_int	ret_wup	loc_cpu	unl_cpu
dis_int	ena_int	chg_ims	ref_ims	

These system calls perform processing that is dependent on the maskable interrupts.
This group provides a function for registering an interrupt handler and subsequently canceling the registration, a function for returning from a interrupt handler, and a function for setting and changing an interrupt-enabled level.

(6) Memory pool management system calls (7)

cre_mpl	del_mpl	get_blk	pget_blk	tget_blk
rel_blk	ref_mpl			

These system calls allocate memory.
This group provides a function for creating and deleting a memory pool, a function for getting and releasing a memory block, and a function for referencing the status of a memory pool.

(7) Time management system calls (6)

set_tim	get_tim	dly_tsk	def_cyc	act_cyc
ref_cyc				

These system calls perform processing that is dependent on time.
This group provides a function for setting or referencing the system clock, a function for placing a task in the timeout wait state, a function for registering a cyclically activated handler and subsequently canceling the registration, and a function for controlling and referencing the state of a cyclically activated handler.

(8) System management system calls (5)

get_ver	ref_sys	def_svc	viss_svc	def_exc

These system calls perform processing that varies with the system.
This group provides a function for obtaining version information, a function for referencing the system status, a function for registering an extended SVC handler and subsequently canceling the registration, and a function for calling an extended SVC handler.

12.2 Calling System Calls

System calls issued from processing programs (tasks/non-tasks) written in C are called as C functions. Their parameters are passed as arguments.

When issuing system calls from processing programs written in assembly language, set parameters and a return address according to the function calling rules of the C compiler, used before calling them with the JAL instruction.

Caution RX4000 declares the prototype of a system call in the stdrx.h file. Accordingly, when issuing a system call from a processing program, the following must be coded to include the header file:

```
#include <stdrx.h>
```

12.3 System Call Function Codes

The system calls supported by RX4000 are assigned function codes conforming to the μITRON3.0 specifications.

Table 12-1 lists the function codes assigned to system codes.

In RX4000, a value of 1 or greater is used when registering an extended SVC handler user described.

Table 12-1. System Call Function Codes

Function code	Classified
–256 to –225	RX4000 original system calls
–224 to –5	System calls conforming to the μITRON3.0 specifications
–4 to 0	Reserved by the system
1 or more	Extended SVC handler

12.4 Data Types of Parameters

The system calls supported by RX4000 have parameters that are defined based on data types that conform to the μITRON3.0 specifications.

Table 12-2 lists the data types of the parameters specified upon the issue of a system call.

Table 12-2. Data Types of Parameters

Macro	Data type	Description
B	char	Signed 8-bit integer
H	short	Signed 16-bit integer
W	long	Signed 32-bit integer
UB	unsigned char	Unsigned 8-bit integer
UH	unsigned short	Unsigned 16-bit integer
UW	unsigned long	Unsigned 32-bit integer
VB	char	Variable data type value (8 bits)
VH	short	Variable data type value (16 bits)
VW	long	Variable data type value (32 bits)
*VP	void	Variable data type value (pointer)
(*FP) ()	void	Program start address
(*VFP) ()	int	Program start address (with return value)
INT	int	Signed 32-bit integer (processor width)
UINT	unsigned int	Unsigned 32-bit integer (processor width)
FN	short	Function code
ID	short	ID number of object
BOOL_ID	short	Pool value or task ID No.
HNO	short	Handler number
ATR	unsigned short	Object attribute
ER	long	Error code
PRI	short	Task priority
TMO	long	Wait time
CYCTIME	long	Cyclically activated time interval (residual time)
DLYTIME	long	Delay time

12.5 Parameter Value Range

Some of the system call parameters supported by RX4000 have a range of permissible values, while others allow the use of only system reserved specific values.

Table 12-3 lists the ranges of parameter values that can be specified upon the issue of a system call.

Table 12-3. Ranges of Parameter Values

Parameter type	Value range
Task ID No.	0x0 to max_cnt (0x7FFF)
Semaphore ID No.	0x0 to max_cnt (0x7FFF)
Event flag ID No.	0x0 to max_cnt (0x7FFF)
Mailbox ID No.	0x0 to max_cnt (0x7FFF)
Memory pool ID No.	0x0 to max_cnt (0x7FFF)
Specification number of cyclically activated handler	0x1 to max_cnt (0x7FFF)
Function code of extended SVC handler	0x1 to max_cnt (0x7FFF)
Interrupt handler specification No.	0x0 to 0x7
Task priority	0x8 to max_cnt (0xFC)
Message priority	0x1 to 0x7FFF
Maximum number of semaphore resources	0x1 to max_cnt (0x7FFF FFFF)
Wait time	–0x1 to 0x7FFF FFFF
Activation time interval of cyclically activated handler	0x1 to 0x7FFF FFFF
Delay time	0x1 to 0x7FFF FFFF
System clock time	0x0 to 0x7FFF FFFF FFFF
Task stack size	0x0 to 0x7FFF FFFF
Memory pool size	0x1 to 0x7FFF FFFF
Memory block size	0x1 to 0x7FFF FFFF

Remarks max_cnt: A value specified during system configuration.
Values in ( ) indicate the maximum value that can be set by the user during configuration.

12.6 System Call Return Values

The system call return values supported by RX4000 are based on the μITRON3.0 specifications.
Table 12-4 lists the system call return values.

Table 12-4. System Call Return Values

Macro	Value	Description
E_OK	0	Normal termination
E_NOMEM	–10	An area for objects cannot be allocated.
E_NOSPT	–11	A system call with the CF not defined, or an unregistered extended SVC handler was called.
E_RSATR	–24	Invalid object attribute specification
E_PAR	–33	Invalid parameter specification
E_ID	–35	Invalid ID number specification
E_NOEXS	–52	No relevant object exists.
E_OBJ	–63	The status of the specified object is invalid.
E_OACV	–66	An unauthorized ID number was specified.
E_CTX	–69	The state in which a system call is issued is invalid.
E_QOVR	–73	A count exceeded 127.
E_DLT	–81	A relevant object was deleted.
E_TMOUT	–85	Timeout
E_RLWAI	–86	A wait state was forcibly canceled by the rel_wai system call.
EV_SIGNAL	–225	A wait state was forcibly canceled by the vsnd_sig system call.

12.7 System Call Extension

RX4000 supports the extension of system calls (functions coded by users are registered in the nucleus as extended system calls).

No limitations are imposed on those functions registered as extended SVC handler; standard system calls (system calls supported by RX4000) can also be included. If, however, standard system calls that can be issued only in the task state are included, the issue state of the extended SVC handler is limited to “issuable only from task.”

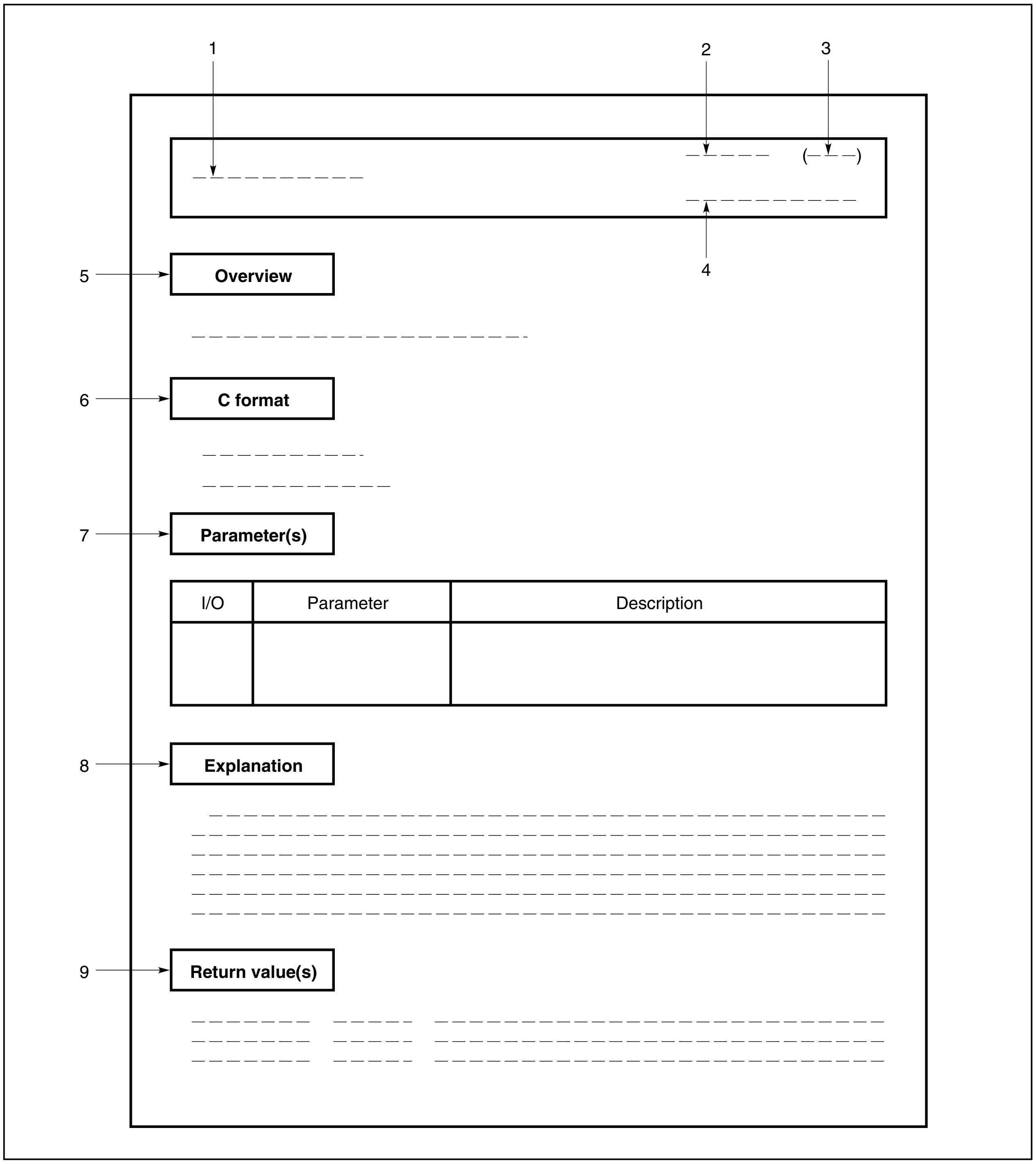
Extended SVC handler are positioned as user-defined system calls, despite their having properties similar to tasks. That is, like standard system calls, the scheduler is started upon the termination of processing and an optimum task is selected.

If a standard system call is included in an extended SVC handler, note that control may pass to another task that is currently processing an extended SVC handler because the scheduler is also started upon the termination of a standard system call.

12.8 Explanation of System Calls

The following explains the system calls supported by RX4000, in the format shown below.

Figure 12-1. System Call Description Format



(1) Name

Indicates the name of the system call.

(2) Semantics

Indicates the source of the name of the system call.

(3) Function code

Indicates the function code of the system call.

(4) Origin of system call

Indicates where the system call can be issued.

Task:	The system call can be issued only from a task.
Non-task:	The system call can be issued only from a non-task (interrupt handler, and cyclically activated handler).
Task/non-task:	The system call can be issued from both a task and non-task.
Task-associated handler	Can be issued from a task-associated handler only.
Interrupt handler:	The system call can be issued only from a interrupt handler.
Cyclically activated handler:	The system call can be issued only from a cyclically activated handler.

(5) Overview

Outlines the functions of the system call.

(6) C format

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

(7) Parameter(s)

System call parameters are explained in the following format.

I/O	Parameter	Description
A	B	C

A: Parameter classification
- I ... Parameter input to RX4000
- O ... Parameter output from RX4000

B: Parameter data type

C: Description of parameter

8 Explanation

Explains the function of a system call.

9 Return value

Indicates a system call's return value using a macro and value.

Return value marked with an asterisk (*):	Value returned by both RX4000 having and that not having the parameter check function
Return value not marked with an asterisk (*):	Value returned only by RX4000 having the parameter check function

12.8.1 Task management system calls

This section explains that group of system calls (task management system calls) that are used to manipulate the task status.

Table 12-5 lists the task management system calls.

Table 12-5. Task Management System Calls

System call	Function
cre_tsk	Generates another task.
del_tsk	Deletes another task.
sta_tsk	Activates another task.
ext_tsk	Terminates the task which issued the system call.
exd_tsk	Terminates the task which issued the system call, then deletes it.
ter_tsk	Forcibly terminates another task.
dis_dsp	Disables dispatch processing.
ena_dsp	Enable dispatch processing.
chg_pri	Change the priority of a task.
rot_rdq	Rotates a task ready queue.
rel_wai	Forcibly releases another task from wait state.
get_tid	Acquires ID number of task which issued the system call.
ref_tsk	Acquires task information.

cre_tsk

Create Task (–17)

Task

Overview

Generates another task.

C format

- When an ID number is specified

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

- When an ID number is not specified

```
#include       <stdrx.h>
ER             ercd = cre_tsk(ID_AUTO, T_CTSK *pk_ctsk, ID *p_tskid);
```

Parameters

I/O	Parameter	Description
I	ID *tskid;*	Task ID number
I	T_CTSK **pk_ctsk;*	Activation address of packet storing the task creation information
O	ID **p_tskid;*	Address of area used to store an ID number

- Structure of task creation information T_CTSK

```
typedef    struct    t_ctsk {
           VP        exinf;      /*  Extended information                           */
           ATR       tskatr;     /*  Task attribute                                 */
           FP        task;       /*  Task activation address                        */
           PRI       itskpri;    /*  Task initial priority (assigned upon activation) */
           INT       stksz;      /*  Task stack size                                */
           VP        gp;         /*  Specific GP register value for task            */
} T_CTSK;
```

Explanation

RX4000 supports two types of interfaces for task creation: one for which an ID number is specified for task creation, and another for which an ID number is not specified.

- When an ID number is specified
  A task having an ID number specified in *tskid* is created based on the information specified in *pk_ctsk*.
  The specified task changes from the non-existent state to the dormant state, in which it is managed by RX4000.

- When an ID number is not specified
  A task is created based on the information specified in *pk_ctsk*.
  The specified task changes from the non_existent state to the dormant state, in which it is managed by RX4000.
  An ID number is allocated by RX4000 and the allocated ID number is stored in the area specified in *p_tskid*.

The following describes task creation information in detail.

exinf ... Extended information

exinf is an area for storing user-specific information on a specified task. It can be used as necessary by the user.

Information set in exinf can be acquired dynamically by issuing the ref_tsk system call from a processing program (task/non-task).

tskatr ... Task attribute

Bit 0 .. Task language

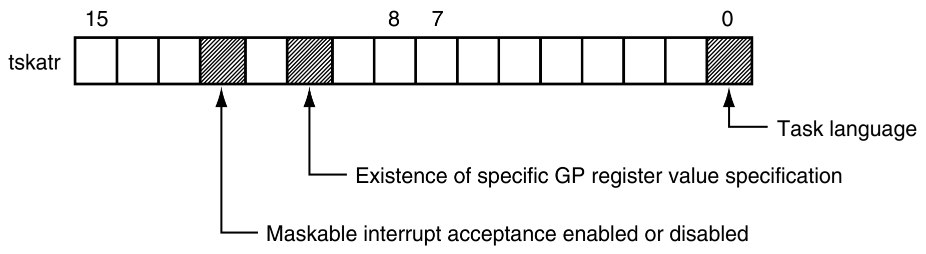
TA_ASM(0):	Assembly language
TA_HLNG(1):	C

Bit 10 .. Existence of specific GP register value specification

TA_DPID(1):	A specific GP register value is specified.

Bit 12 .. Maskable interrupt acceptance enabled or disabled

TA_ENAINT(0):	When a task is activated, the acceptance of maskable interrupts is enabled.
TA_DISINT(1):	When a task is activated, the acceptance of maskable interrupts is disabled.


15
8
7
0
tskatr
Task language
Existence of specific GP register value specification
Maskable interrupt acceptance enabled or disabled

task	...	Task activation address
itskpri	...	Task initial priority (assigned upon activation)
stksz	...	Stack size of task (bytes)
gp	...	Specific GP register value for task

Remark When the value 1 of bit 10 of tskatr is other than TA_DPID, the contents of gp are meaningless.

Return value

*E_OK	0	Normal termination
*E_NOMEM	–10	An area for task management block cannot be allocated.
E_RSATR	–24	Invalid specification of attribute tskatr
E_PAR	–33	Invalid parameter specification – The start address of packet storing task creation information is invalid (*pk_ctsk* = 0). – Invalid activation address specification (task = 0) – Invalid initial priority specification (itskpri ≤ 0, maximum priority < itskpri) – The address of the area used to store an ID number is invalid (*p_tskid* = 0) (When a task is created with no ID number specified)
E_ID	–35	Invalid ID number specification (maximum number of tasks created < *tskid*)
*E_OBJ	–63	A task having a specified ID number is created.
E_OACV	–66	An unauthorized ID number (*tskid* ≤ 0) was specified.
E_CTX	–69	The cre_tsk system call was issued from a non-task.

del_tsk

Delete Task (–18)

Task

Overview

Deletes another task.

C format

```
#include      <stdrx.h>
ER            ercd = del_tsk(ID tskid);
```

Parameter

I/O	Parameter	Description
I	ID *tskid;*	Task ID number

Explanation

This system call changes the task specified in *tskid* from the dormant state to the non-existence state. This releases the task from the control of RX4000.

Furthermore, if delete your own task, issue the exd_tsk system call.

Caution This system call does not queue delete requests. Accordingly, if a specified task is not in the dormant state, this system call returns E_OBJ as a return value.

Return value

*E_OK	0	Normal termination
E_ID	–35	Invalid ID number specification (maximum number of tasks created < *tskid*)
*E_NOEXS	–52	The specified task does not exist.
*E_OBJ	–63	The specified task is not in the dormant state.
E_OACV	–66	An unauthorized ID number (*tskid* ≤ 0) was specified.
E_CTX	–69	The del_tsk system call was issued from a non-task.

sta_tsk

Start Task (–23)

Task/nontask

Overview

Activates another task.

C format

```
#include      <stdrx.h>
ER            ercd = sta_tsk(ID tskid, INT stacd);
```

Parameters

I/O	Parameter	Description
I	ID *tskid;*	Task ID number
I	INT *stacd*;	Activation code

Explanation

This system call changes the task specified with *tskid* from the dormant state to the ready state.

The specified task is scheduled by RX4000.

For *stacd*, specify the activation code to be passed to the specified task. The specified task can be manipulated by handling the activation code as if it were a function parameter.

Caution This system call does not queue activation requests. Accordingly, when a specified task is not in the dormant state, this system call returns E_OBJ as the return value.

Return value

*E_OK	0	Normal termination
E_ID	–35	Invalid ID number specification (maximum number of tasks created < *tskid*)
*E_NOEXS	–52	The specified task does not exist.
*E_OBJ	–63	The specified task is not in the dormant state.
E_OACV	–66	An unauthorized ID number (*tskid* ≤ 0) was specified.

Exit Task (–21)

ext_tsk

Task

Overview

Terminates the task which issued the system call.

C format

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

Parameter

None.

Explanation

This system call changes the state of the task from the run state to the dormant state.
The task is excluded from RX4000 scheduling.

Remark 1. This system call initializes "task creation information", specified at task creation (at configuration or upon the issue of a cre_tsk system call).

2. If a task is coded in assembly language, code the following to terminate the task.

```
jr              _ext_tsk
```

Cautions 1. If this system call is issued from a non-task or in the dispatch disabled state, its operation is not guaranteed.

2. This system call does not release those resources (semaphore count, memory block, etc.) that were acquired before the termination of the task. Accordingly, the user has to release such resources before issuing this system call.

Return value

None.

exd_tsk

Exit and DeleteTask (–22)

Task

Return value

Terminates the task which issued the system call, then deletes it.

C format

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

Parameter

None.

Explanation

This system call changes the task from the run state to the non-existent state.
This releases the task from the control of RX4000.

Remark If a task is coded in assembly language, perform coding as follows to terminate the task:

```
jr          _ext_tsk
```

Cautions 1. If this system call is issued from a non-task or in the dispatch disabled state, its operation is not guaranteed.

2. This system call does not release those resources (semaphore count, memory block, etc.) that were acquired before the termination of the task. Accordingly, the user has to release such resources before issuing this system call.

Return value

None.

ter_tsk

Terminate Task (–25)

Task

Overview

Forcibly terminates another task.

C format

```
#include      <stdrx.h>
ER            ercd = ter_tsk(ID tskid);
```

Parameter

I/O	Parameter	Description
I	ID *tskid;*	Task ID number

Explanation

This system call forcibly changes the state of the task specified in *tskid* to the dormant state.

However, if a task-associated handler is registered in the object task and the tasks force end processing is not masked, the task-associated handler will start. The handler will operate as a part of the task, but it has a higher priority level than other tasks.

Remark This system call initializes the "task creation information" specified at task creation (at configuration or upon the issue of a cre_tsk system call).

Cautions 1. This system call does not queue termination requests. Accordingly, if a specified task is not in the ready, wait, suspend, or wait_suspend state, this system call returns E_NOEXS or E_OBJ as the return value.

2. This system call does not release those resources (semaphore count, memory block, etc.) that were acquired before the termination of the specified task. Therefore, release such resources at the user side or within the task-associated handler before issuing this system call.

Return value

*E_OK	0	Normal termination
E_ID	–35	Invalid ID number specification (maximum number of tasks created < *tskid*)
*E_NOEXS	–52	The specified task does not exist.
*E_OBJ	–63	The specified task is that task which issued this system call, or the task is in the dormant state.
E_OACV	–66	An unauthorized ID number (*tskid* ≤ 0) was specified.
E_CTX	–69	The ter_tsk system call was issued from a non-task.

dis_dsp

Disable Dispatch (–30)

Task

Overview

Disables dispatch processing.

C format

```
#include      <stdrx.h>
ER            ercd = dis_dsp(void);
```

Parameter

None.

Explanation

This system call disables dispatch processing (task scheduling).
Dispatch processing is disabled until the ena_dsp system call is issued after this system call has been issued.
If a system call such as chg_pri or sig_sem is issued to schedule tasks after the dis_dsp system call is issued but before the ena_dsp system call is issued, RX4000 merely performs operations on a queue and delays actual scheduling until the ena_dsp system call is issued, at which time the processing is performed at one time.

Cautions 1. This system call does not queue disable requests. Accordingly, if the dis_dsp system call has already been issued and dispatch processing has been disabled, no processing is performed and a disable request is not handled as an error.

2. If a system call such as wait_sem and wai_flg is issued, causing the state of the task to change to the wait state after the dis_dsp system call is issued but before the ena_dsp system call is issued, RX4000 returns E_CTX as the return value, regardless of whether the wait conditions are satisfied.

Return value

*E_OK	0	Normal termination
*E_CTX	–69	Context error – The dis_dsp system call was issued from a non-task. – The dis_dsp system call was issued after the loc_cpu system call was issued.

ena_dsp

Enable Dispatch (–29)

Task

Overview

Enable dispatch processing.

C format

```
#include      <stdrx.h>
ER            ercd = ena_dsp(void);
```

Parameter

None.

Explanation

This system call enables dispatch processing (task scheduling).
If a system call such as chg_pri and sig_sem is issued to schedule tasks after the dis_dsp system call is issued but before the ena_dsp system call is issued, RX4000 merely performs operations on a queue and delays actual scheduling until the ena_dsp system call is issued, at which time the processing is performed at one time.

Caution This system call does not queue resume requests. Accordingly, if the ena_dsp system call has already been issued and dispatch processing has been resumed, no processing is performed. The resume request is not handled as an error.

Return value

*E_OK	0	Normal termination
*E_CTX	–69	Context error – The ena_dsp system call was issued from a non-task. – The ena_dsp system call was issued after the loc_cpu system call had been issued.

chg_pri

Change Priority (–27)

Task/nontask

Overview

Changes the priority of a task.

C format

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

Parameters

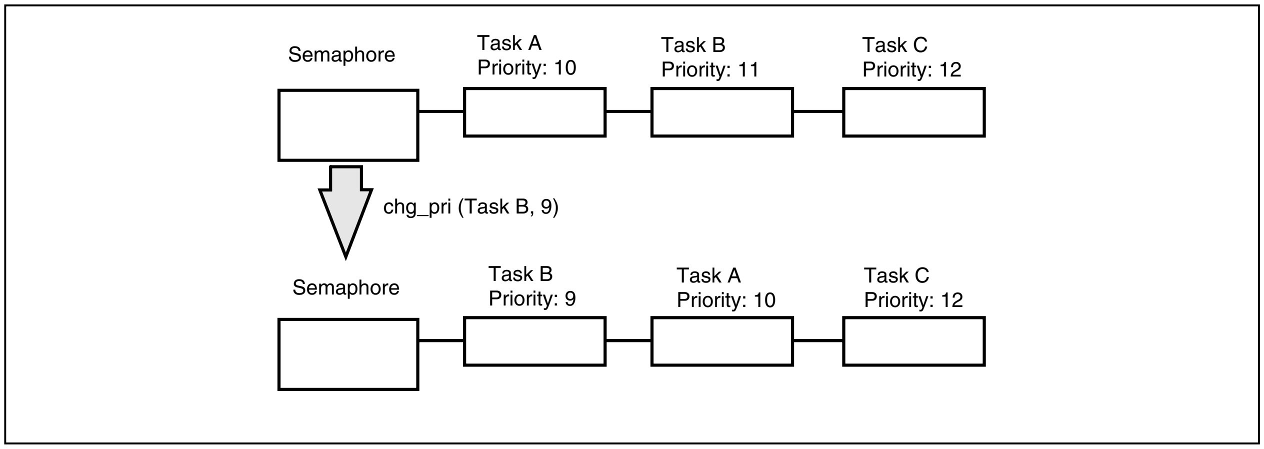
I/O	Parameter	Description
I	ID *tskid;*	Task ID number TSK_SELF(0): Task which issued this system call Value: Task ID number
I	PRI *tskpri;*	Task priority TPRI_INI(0): Task initial priority Value: Task priority

Explanation

This system call changes the value of the task priority specified in *tskid* to that specified in *tskpri*.
If the object task is in the run state or the ready state, this system call executes priority change processing and queues the object task at the tail end of the ready queue in accordance with its priority.

Remarks 1. If a specified task is placed in a queue according to its priority, the issue of the chg_pri system call may change the wait order.

Example When three tasks (task A: priority 10, task B: priority 11, task C: priority 12) are placed in a semaphore queue according to their priority, and if the priority of task B is changed from 11 to 9, then the wait order of the queue changes as shown below.



Remarks 2. The value specified by *tskpri* is active until the next chg_pri system call is issued, or until the object task changes to the dormant state.

3. The task priority in RX4000 becomes higher as its value decreases.

Return value

*E_OK	0	Normal termination
E_PAR	–33	Invalid priority specification (*tskpri* < 0, maximum priority < *tskpri*)
E_ID	–35	Invalid ID number specification – maximum number of tasks created < *tskid* – When the chg_pri system call was issued from a non-task, TSK_SELF was specified in *tskid*.
*E_NOEXS	–52	The specified task does not exist.
*E_OBJ	–63	The specified task is in the dormant state.
E_OACV	–66	An unauthorized ID number (*tskid* < 0) was specified.

rot_rdq

Rotate Ready Queue (–28)

Task/nontask

Overview

Rotates a task ready queue.

C format

```
#include      <stdrx.h>
ER            ercd = rot_rdq(PRI tskpri);
```

Parameter

I/O	Parameter	Description
I	PRI *tskpri;*	Task priority TPRI_RUN(0): Priority of task in run state Value: Task priority

Explanation

This system call queues the first task in a ready queue to the end of the queue according to the priority specified in *tskpri.*

Notes 1. If no task of a specified priority exists in a ready queue, this system call performs no processing. This is not regarded as an error.

2. By issuing the rot_rdq system call at regular intervals, round-robin scheduling can be achieved.

Return value

*E_OK	0	Normal termination
E_PAR	–33	Invalid priority specification (*tskpri* < 0, maximum priority < *tskpri*)

rel_wai

Release Wait (–31)

Task/nontask

Overview

Forcibly releases another task from the wait state.

C format

```
#include      <stdrx.h>
ER            ercd = rel_wai(ID tskid);
```

Parameter

I/O	Parameter	Description
I	ID *tskid;*	Task ID number

Explanation

This system call forcibly releases the task, specified in *tskid*, from the wait state.
The specified task is excluded from a queue, and its states changes from the wait state to the ready state, or from the wait_suspend state to the suspend state.
For a task released from the wait state by the rel_wai system call, E_RLWAI is returned as the return value of the system call (slp_tsk, wai_sem, etc.) that caused transition to the wait state.

Caution The rel_wai system call does not release the suspend state.

Return value

*E_OK	0	Normal termination
E_ID	–35	Invalid ID number specification (maximum number of tasks created < *tskid*)
*E_NOEXS	–52	The specified task does not exist.
*E_OBJ	–63	The specified task is in neither the wait nor wait_suspend state.
E_OACV	–66	An unauthorized ID number (*tskid* ≤ 0) was specified.

get_tid

Get Task Identifier (–24)

Task/nontask

Overview

Acquires a task ID number.

C format

```
#include       <stdrx.h>
ER             ercd = get_tid(ID *p_tskid);
```

Parameter

I/O	Parameter	Description
O	ID *p_tskid;*	Address of an area used to store an ID number

Explanation

This system call stores, in the area specified in *p_tskid*, the ID number of the task which issued this system call.

Caution If this system call is issued from a non-task, FALSE (0) is stored in the area specified in *p_tskid*.

Return value

*E_OK	0	Normal termination
E_PAR	–33	The address of the area used to store an ID number is invalid (*p_tskid* = 0).

Refer Task Status (–20)

ref_tsk

Task/nontask

Overview

Acquires task information.

C format

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

Parameters

I/O	Parameter	Description
O	T_RTSK *pk_rtsk;*	Start address of packet used to store task information
I	ID *tskid;*	Task ID number TSK_SELF(0): Task which issued this system call Value: Task ID number

• Structure of task information T_RTSK

```
typedef   struct   t_rtsk {
          VP       exinf;         /*  Extended information         */
          PRI      tskpri;        /*  Current priority             */
          UNIT     taskstat;      /*  Task status                  */
          UNIT     tskwait;       /*  Wait cause                   */
          ID       wid;           /*  ID number of wait object     */
          INT      wupcnt;        /*  Number of wake-up requests   */
          INT      suscnt;        /*  Number of suspend requests   */
} T_RTSK;
```

Explanation

This system call stores the task information (extended information, current priority, etc.) specified in *tskid* in the packet specified in *pk_rtsk*.
The following describes the task information in detail.

exinf	...	Extended information
tskpri	...	Current priority

tskstat	...	Task state	
		TTS_RUN(H’01):	run state
		TTS_RDY(H’02):	ready state
		TTS_WAI(H’04):	wait state
		TTS_SUS(H’08):	suspend state
		TTS_WAS(H’0c):	wait_suspend state
		TTS_DMT(H’10):	dormant state
tskwait	...	Type of wait state	
		TTW_SLP(H’0001):	Wake-up wait state
		TTW_DLY(H’0002):	Timeout wait state
		TTW_FLG(H’0010):	Event flag wait state
		TTW_SEM(H’0020):	Resource wait state
		TTW_MBX(H’0040):	Message wait state
		TTW_MPL(H’1000):	Memory block wait state
wid	...	ID number of wait object (semaphore, event, flag, etc.)	
wupcnt	...	Number of wake-up requests	
suscnt	...	Number of suspend requests	

Remarks 1. When the value of tskstat is other than TTS_WAI or TTS_WAS, the contents of tskwait will be undefined.

2. When the value of tskwait is other than TTW_FLG, TTW_SEM, TTW_MBX, or TTW_MPF, the contents of wid will be undefined.

Return value

*E_OK	0	Normal termination
E_PAR	–33	The start address of the packet used to store task information is invalid (*pk_rtsk* = 0)
E_ID	–35	Invalid ID number specification – Maximum number of tasks created < *tskid* – When the ref_tsk system call was issued from a non-task, TSK_SELF was specified in *tskid*.
*E_NOEXS	–52	The specified task does not exist.
E_OACV	–66	An unauthorized ID number (*tskid* < 0) was specified.

12.8.2 Task-associated handler function system calls

This section explains that group of system call (task-associated handler function system calls) that are associated to tasks and perform handler operations.

Table 12-6 lists the task-associated handler function system calls.

Table 12-6. Task-Associated Handler Function System Calls

System call	Function
vdef_sig	Registers and cancels registration of task-associated handlers.
vsnd_sig	Sends signals.
vchg_sms	Changes signal masks.
vref_sms	Acquires signal masks.

vdef_sig

Define Signal Handler (–241)

Task

Overview

Registers and cancels registration of task-associated handlers.

C format

```
#include        <stdrx.h>
ER              ercd = vdef_sig(T_CMPL *pk_dsig);
```

Parameters

I/O	Parameter	Description
O	T_DSIG *pk_dsig;	Start address of the packet where the task-associated handler's registration information is stored.

- Structure of task-associated handler registration information T_DSIG

```
typedef   struct   t_dsig {
          ATR      sigatr;    /* Task-associated handler attribute                  */
          FP       sighdr;    /* Task-associated handler activate address           */
          UINT     isigms;    /* Signal mask value during task-associated handler   */
                                 registration
          VP       gp;        /* Specific GP register value for task-associated handler */
} T_DSIG;
```

Explanation

Registers a task-associated handler which starts when an external occurrence (signal) is communicated to the task based on the information specified by pk_dsig.
Detailed task-associated handler registration information is shown below.

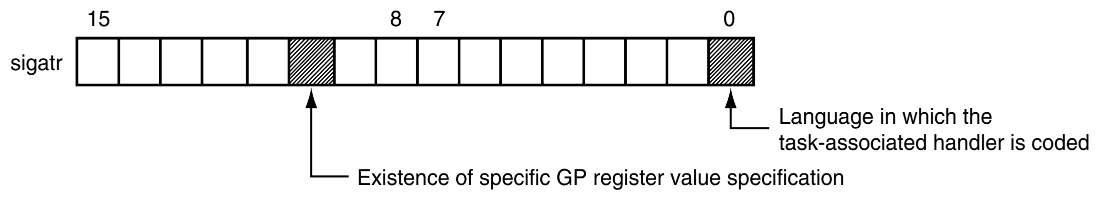
sigatr ... Attribute of task-associated handler

Bit 0 .. Language in which the task-associated handler is coded

TA_ASM(0):	Assembly language
TA_HLNG(1):	C

Bit 10 .. Existence of specific GP register value specification

TA_PID(1):	Specifies a specific GP register value


15
8
7
0
sigatr
Language in which the task-associated handler is coded
Existence of specific GP register value specification

sighdr	...	Task-associated handler activate address
isigms	...	Signal mask value during task-associated handler registration
gp	...	Specific GP register value for task-associated handler

If a task-associated handler is already registered when this system call is registered, it is not treated as an error, but the task-associated handler specified by this system call is newly registered.

Also, if NADR (-1) is set in the area specified by pk_dsig when this system call is issued, the task-associated handler's registeration is canceled.

Remark When the value 1 of bit 10 of sigatr is other than TA_PID, the contents of gp are meaningless.

Return value

*E_OK	0	Normal termination
E_RSATR	–24	Invalid specification of attribute sigatr.
E_PAR	–33	Invalid parameter specification. – The start address of the packet storing task-associated handler registration information is invalid (pk_dsig = 0). – Invalid activation address specification (sighdr = 0).
E_CTX	–69	The vdef_sig system call was issued from a non-task.

vsnd_sig

Send Signal (–247)

Task/nontask

Overview

Sends signals.

C format

```
#include      <stdrx.h>
ER            ercd = vsnd_sig(ID tskid, UINT ssigno);
```

Parameter

I/O	Parameter		Description
I	ID	*tskid;*	Task ID number
I	UINT	*ssigno;*	Signal number of the sending signal.

Explanation

This sends the signals specified by *ssigno* to the task specified by *tskid*. If a task-associated handler is defined in the object task, and if the specified signal is not masked, the task-associated handler activates.
In this way, the object task-associated handler becomes an object of RX4000 scheduling. Also, if the object task is in the wait state at this time (resources wait state, message wait state, etc.), it is forcibly released.
An EV_SIGNAL is returned to a task for which the wait state (resources wait state, message wait state, etc.) was released by issue of the vsnd_sig system call as the return value of a system call (wai_sem, rcv_msg, etc.) which becomes an opportunity for that task to change to the wait state again.
The object task-associated handler can operate handling *ssigno* in the same way as a function parameter. This parameter can be used to judge signals in the task-associated handler.

Return value

*E_OK	0	Normal termination
E_PAR	–33	Invalid signal number specification (*ssigno*<4, 7< *ssigno*, *ssigno*<12, 31< *ssigno*)
E_ID	–35	Invalid ID number specification (maximum number of tasks created < *tskid*).
*E_NOEXS	–52	The specified task does not exist.
*E_OBJ	–63	The specified task is in the dormant state.
E_OACV	–66	An unauthorized ID number (*tskid* ≤ 0) was specified.

Change Signal Mask (–243)

vchg_sms

Task

Overview

Changes the signal mask.

C format

```
#include      <stdrx.h>
ER            ercd = vchg_sms(UINT sigms);
```

Parameter

I/O	Parameter	Description
I	UINT *sigms;*	Signal mask

Explanation

Changes the signal mask of the task to the signal mask specified by *sigms*.
In RX4000, there are 32 levels of signal factors and enable or disable can be specified for each respective level. If each signal is enabled, 0 is set and if each is disabled, 1 is set.

Return value

*E_OK	0	Normal termination
E_CTX	–69	The vchg_sms system call was issued from a non-task

vref_sms

Refer Signal Mask (–244)

Task

Overview

Acquires a signal mask.

C format

```
#include        <stdrx.h>
ER              ercd = vref_sms(UINT *p_sigms);
```

Parameter

I/O	Parameter	Description
I	UINT **p_sigms;*	Address of the area where the signal mask is stored.

Explanation

Stores the signal mask of its own task in the address specified by *p_sigms*.

Return value

*E_OK	0	Normal termination
E_PAR	–33	The address of the area used to store a signal mask is invalid (p_sigms = 0).
E_CTX	–69	The vref_sms system call was issued from a non-task.

12.8.3 Task-associated synchronization system calls

This section explains a group of system calls (task-associated synchronization system calls) that perform the synchronous operations associated with tasks.

Table 12-7 lists the task-associated synchronization system calls.

Table 12-7. Task-Associated Synchronization System Calls

System call	Function
sus_tsk	Places another task in the suspend state.
rsm_tsk	Resumes a task in the suspend state.
frsm_tsk	Forcibly resumes a task in the suspend state.
slp_tsk	Places the task which issued this system macro into the wake-up wait state.
tslp_tsk	Places the task which issued this system macro (with timeout) into the wake-up wait state.
wup_tsk	Wakes up another task.
can_wup	Cancels a request to wake up a task.

sus_tsk

Suspend Task (–33)

Task/nontask

Overview

Places another task in the suspend state.

C format

```
#include      <stdrx.h>
ER            ercd = sus_tsk(ID tskid);
```

Parameter

I/O	Parameter	Description
I	ID *tskid;*	Task ID number

Explanation

This system call issues a suspend request to the task specified in *tskid* (the suspend request counter is incremented by 0x1).
If a specified task is in the ready or wait state when this system call is issued, this system call changes the specified task from the ready state to the suspend state or from the wait state to the wait_suspend state, and also issues a suspend request (increments the suspend request counter).

Caution The suspend request counter managed by RX4000 consists of seven bits. Therefore, once the number of suspend requests exceeds 127, the sus_tsk system call returns E_QOVR as a return value without incrementing the suspend request counter.

Return value

*E_OK	0	Normal termination
E_ID	–35	Invalid ID number specification (maximum number of tasks created < *tskid*)
*E_NOEXS	–52	The specified task does not exist.
*E_OBJ	–63	The specified task is in the dormant state, or the task is its own task.
E_OACV	–66	An unauthorized ID number (*tskid* ≤ 0) was specified.
*E_QOVR	–73	The number of suspend requests exceeded 127.

rsm_tsk

Resume Task (–35)

Task/nontask

Overview

Resumes a task in the suspend state.

C format

```
#include      <stdrx.h>
ER            ercd = rsm_tsk(ID tskid);
```

Parameter

I/O	Parameter	Description
I	ID *tskid;*	Task ID number

Explanation

This system call cancels only one of the suspend requests that are issued to the task specified in *tskid* (the suspend request counter is decremented by 0x1).
If the issue of this system call causes the suspend request counter for the specified task to be 0x0, this system call changes the task from the suspend state to the ready state or from the wait_suspend state to the wait state.

Caution This system call does not queue cancel requests. Accordingly, if a specified task is not in the suspend or wait_suspend state, this system call returns E_OBJ as a return value without decrementing a suspend request counter.

Return value

*E_OK	0	Normal termination
E_ID	–35	Invalid ID number specification (maximum number of tasks created < *tskid*)
*E_NOEXS	–52	The specified task does not exist.
*E_OBJ	–63	The specified task is in the suspend or wait_suspend state.
E_OACV	–66	An unauthorized ID number (*tskid* ≤ 0) was specified.

frsm_tsk

Force Resume Task (–36)

Task/nontask

Overview

Forcibly resumes a task in the suspend state.

C format

```
#include        <stdrx.h>
ER              ercd = frsm_tsk(ID tskid);
```

Parameter

I/O	Parameter	Description
I	ID *tskid;*	Task ID number

Explanation

This system call cancels all the suspend requests issued to the task specified in *tskid* (the suspend request counter is set to 0x0).

The specified task changes from the suspend state to the read state or from the wait_suspend state to the wait state.

Caution This system call does not queue cancel requests. Accordingly, if a specified task is in neither the suspend or wait_suspend state, this system call returns E_OBJ as the return value without setting the suspend request counter.

Return value

*E_OK	0	Normal termination
E_ID	–35	Invalid ID number specification (maximum number of tasks created < *tskid*)
*E_NOEXS	–52	The specified task does not exist.
*E_OBJ	–63	The specified task is in the suspend or wait_suspend state.
E_OACV	–66	An unauthorized ID number (*tskid* $\leq$ 0) was specified.

slp_tsk

Sleep Task (–38)

Task

Overview

Places the task which issued this system macro into the wake-up wait state.

C format

```
#include      <stdrx.h>
ER            ercd = slp_tsk(void);
```

Parameter

None.

Explanation

This system call cancels only one of the wake-up requests issued to the task (the wake-up request counter is decremented by 0x1).

If the wake-up request counter for the task is 0x0 when this system call is issued, this system call changes the state of the task from the run state to the wait state (wake-up wait state) without canceling a wake-up request (the wake-up request counter is decremented).

The wake-up wait state is released when a wup_tsk, ret_wup, or rel_wai system call is issued. The task changes from the wake-up wait state to the ready state.

Return value

*E_OK	0	Normal termination
E_CTX	–69	Context error – The slp_tsk system call was issued from a non-task. – The slp_tsk system call was issued in the dispatch disabled state.
*E_RLWAI	–86	The wake-up wait state was forcibly released by the rel_wai system call.
EV_SIGNAL	–225	The wake-up wait state was forcibly released by the vsnd_sig system call.

tslp_task

Sleep Task with Timeout (–37)

Task

Overview

Places the task which issued this system macro (with timeout) into the wake-up wait state.

C format

```
#include      <stdrx.h>
ER            ercd = tslp_tsk(TMO tmout);
```

Parameter

I/O	Parameter		Description	
I	TMO	*tmount;*	Wait time	
			TMO_POL(0)	Quick return
			TMO_FEVR(–1):	Permanent wait
			Value:	Wait time

Explanation

This system call cancels only one of the wake-up requests issued to the task (the wake-up request counter is decremented by 0x1).

If the wake-up request counter for the task is 0x0 when this system call is issued, this system call changes the task from the run state to the wait state (wake-up wait state) without canceling a wake-up request (the wake-up request counter is decremented).

Furthermore, the wake_up wait state is canceled if the wait time specified by tmout passes or if the wup_tsk, ret_wup, or rel_wai system call is issued, and its own task changes to the ready state.

Return value

*E_OK	0	Normal termination
E_PAR	–33	Invalid wait time specification (*tmout* < TMO_FEVR)
E_CTX	–69	Context error – This system call was issued from a non-task. – This system call was issued in the dispatch disabled state.
*E_TMOUT	–85	The wait time has elapsed.
*E_RLWAI	–86	The wake-up wait state was forcibly released by a rel_wai system call.
EV_SIGNAL	–225	The wake-up wait state was forcibly released by a vsnd_sig system call.

wup_tsk

Wakeup Task (–39)

Task/nontask

Overview

Wakes up another task.

C format

```
#include      <stdrx.h>
ER            ercd = wup_tsk(ID tskid);
```

Parameter

I/O	Parameter	Description
I	ID *tskid;*	Task ID number

Explanation

This system call issues a wake-up request to the task specified in *tskid* (the wake-up request counter is incremented by 0x1).
If the specified task is in the wait state (wake-up wait state) when this system call is issued, this system call changes the task from the wake-up wait state to the ready state without issuing a wake-up request (the wake-up request counter is incremented).

Caution A wake-up request counter managed by RX4000 consists of 7-bit width. Therefore, when the number of wake-up requests exceeds 127, the wup_tsk system call returns E_QOVR as the return value without incrementing the wake-up request counter.

Return value

*E_OK	0	Normal termination
E_ID	–35	Invalid ID number specification (maximum number of tasks created < *tskid*)
*E_NOEXS	–52	The specified task does not exist.
*E_OBJ	–63	The specified task is in the dormant state, or the task is its own task.
E_OACV	–66	An unauthorized ID number (*tskid* ≤ 0) was specified.
*E_QOVR	–73	The number of wake-up requests exceeded 127.
EV_SIGNAL	–225	The wake-up wait state was forcibly released by a vsnd_sig system call.

can_wup

Cancel Wakeup Task (–40)

Task/nontask

Overview

Cancels a request to wake up a task.

C format

```
#include        <stdrx.h>
ER              ercd = can_wup(INT *p_wupcnt, ID tskid);
```

Parameter

I/O	Parameter		Description
O	INT	**p_wupcnt;*	Address of area used to store the number of wake-up requests
I	ID	*tskid;*	Task ID number TSK_SELF(0): Local task Value: Task ID number

Explanation

This system call cancels all the wake-up requests issued to the task specified in *tskid* (the wake-up request counter is set to 0x0).
The number of wake-up requests canceled by this system call is stored in the area specified in *p_wupcnt*.

Return value

*E_OK	0	Normal termination
E_PAR	–33	The address of the area used to store the number of wake-up requests is invalid (*p_wupcnt* = 0)
E_ID	–35	Invalid ID number specification – Maximum number of tasks created < *tskid* – When the can_wup system call was issued from a non-task, TSK_SELF was specified in *tskid*.
*E_NOEXS	–52	The specified task does not exist.
*E_OBJ	–63	The specified task is in the dormant state.
E_OACV	–66	An unauthorized ID number (*tskid* < 0) was specified.

12.8.4 Synchronous communication system calls

This section explains those system calls that are used for synchronization (exclusive control and queuing) and communication between tasks.

Table 12-8 lists the synchronous communication system calls.

Table 12-8. Synchronous Communication System Calls

System call	Function
cre_sem	Generates a semaphore.
del_sem	Deletes a semaphore.
sig_sem	Returns resources.
wai_sem	Acquires resources.
preq_sem	Acquires resources (polling).
twai_sem	Acquires resources (with timeout).
ref_sem	Acquires semaphore information.
cre_flg	Creates an event flag.
del_flg	Deletes an event flag.
set_flg	Sets a bit pattern.
clr_flg	Clears a bit pattern.
wai_flg	Checks a bit pattern.
pol_flg	Checks a bit pattern (polling).
twai_flg	Checks a bit pattern (with timeout).
ref_flg	Acquires event flag information.
cre_mbx	Creates a mailbox.
del_mbx	Deletes a mailbox.
snd_msg	Sends a message.
rcv_msg	Receives a message.
prcv_msg	Receives a message (polling).
trcv_msg	Receives a message (with timeout).
ref_mbx	Acquires mailbox information.

cre_sem

Create Semaphore (–49)

Task

Overview

Generates a semaphore.

C format

- When an ID number is specified

```
#include      <stdrx.h>
ER            ercd = cre_sem(ID semid, T_CSEM *pk_csem);
```

- When an ID number is not specified

```
#include      <stdrx.h>
ER            ercd = cre_sem(ID_AUTO, T_CSEM *pk_csem, ID *p_semid);
```

Parameter

I/O	Parameter	Description
I	ID *semid;*	Semaphore ID number
I	T_CSEM **pk_csem;*	Start address of packet containing semaphore creation information.
O	ID **p_semid;*	Address of area used to store an ID number

- Structure of semaphore creation information T_CSEM

```
typedef    struct    t_csem {
           VP        exinf;      /*  Extended information                 */
           ATR       sematr;     /*  Semaphore attribute                  */
           INT       isemcnt;    /*  Initial semaphore resource count     */
           INT       maxsem;     /*  Maximum semaphore resource count     */
} T_CSEM;
```

Explanation

RX4000 provides two types of interfaces for semaphore creation: one where an ID number must be specified, and another where the ID number is not required.

- When an ID number is specified
  A semaphore having an ID number specified in *semid* is created based on the information specified in *pk_csem*.

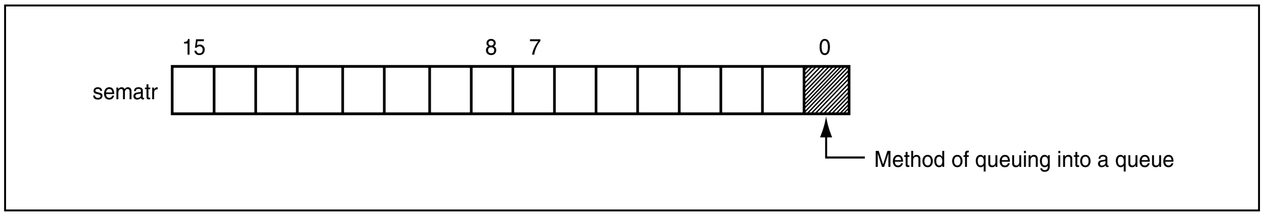
- When an ID number is not specified
  A semaphore is created based on the information specified in *pk_csem*.
  An ID number is allocated by RX4000 and the allocated ID number is stored into the area specified with *p_semid*.

The following describes semaphore creation information in detail.

exinf	...	Extended information An area for storing user-specific information on a specified semaphore. The user can use this area as required. Information set in exinf can be dynamically acquired by issuing the ref_sem system call from a processing program (tasks and non-tasks).
sematr	...	Semaphore attribute

Bit 0 .. Method of queuing into a queue

TA_TPRI(0):	Priority order
TA_TFIFO(1):	FIFO order



isemcnt	...	Initial semaphore resource count
maxsem	...	Maximum semaphore resource count

Return value

*E_OK	0	Normal termination
E_RSATR	–24	Invalid specification of attribute sematr.
E_PAR	–33	Invalid parameter specification – The start address of a packet storing semaphore creation information is invalid (*pk_csem* = 0). – The initial resource count is invalid (isemcnt < 0). – The maximum resource count is invalid (maxsem ≤ 0, maxsem < isemcnt). – The address of the area used to store an ID number is invalid (*p_semid* = 0). (When a semaphore is created without an ID number specified)
E_ID	–35	Invalid ID number specification (maximum number of semaphores created< *semid*)
*E_OBJ	–63	A task having the specified ID number has already been created.
E_OACV	–66	An unauthorized ID number (*semid* ≤ 0) was specified.
E_CTX	–69	The cre_sem system call was issued from a non-task.

del_sem

Delete Semaphore (–50)

Task

Overview

Deletes a semaphore.

C format

```
#include      <stdrx.h>
ER            ercd = del_sem(ID semid);
```

Parameter

I/O	Parameter	Description
I	ID *semid;*	Semaphore ID number

Explanation

This system call deletes the semaphore specified in *semid*.
The specified semaphore is released from RX4000 control.
The task released from the wait state (resource wait state) by the del_sem system call has E_DLT returned as the return value of the system call (wai_sem or twai_sem) that initiated transition to the wait state.

Return value

*E_OK	0	Normal termination
E_ID	–35	Invalid ID number specification (maximum number of semaphores created < *semid*)
*E_NOEXS	–52	The specified semaphore does not exist.
E_OACV	–66	An unauthorized ID number (*semid* ≤ 0) was specified.
E_CTX	–69	The del_sem system call was issued from a non-task.

sig_sem

Signal Semaphore (–55)

Task/nontask

Overview

Returns resources.

C format

```
#include      <stdrx.h>
ER            ercd = sig_sem(ID semid);
```

Parameter

I/O	Parameter	Description
I	ID *semid;*	Semaphore ID number

Explanation

This system call returns resources to the semaphore specified in *semid* (the semaphore counter is incremented by 0x1).

If tasks are queued in the queue of the specified semaphore when this system call is issued, this system call passes the resources to the relevant task (the first task in the queue) without returning the resources (incrementing the semaphore counter).

Consequently, the relevant task is removed from the queue, and its state changes from the wait state (resource wait state) to the ready state, or from the wait_suspend state to the suspend state.

Caution The semaphore counter managed by RX4000 counts up to the maximum number of resources that can be acquired as specified at the time it is generated. Therefore, when the number of resources exceeds the maximum number of resources, by issuing sig_sem system call, the sig_sem system call returns E_QOVR as its return value without incrementing the semaphore counter.

Return value

*E_OK	0	Normal termination
E_ID	–35	Invalid ID number specification (maximum number of semaphores created < *semid*)
*E_NOEXS	–52	The specified semaphore does not exist.
E_OACV	–66	An unauthorized ID number (*semid* ≤ 0) was specified.
*E_QOVR	–73	The resource count exceeded the maximum resource count specified at generation.

wai_sem

Wait on Semaphore (–53)

Task

Overview

Acquires resources.

C format

```
#include      <stdrx.h>
ER            ercd = wai_sem(ID semid);
```

Parameter

I/O	Parameter	Description
I	ID *semid;*	Semaphore ID number

Explanation

This system call acquires resources from the semaphore specified in *semid* (the semaphore counter is decremented by 0x1).
When this system call is issued, if no resource can be acquired from a specified semaphore (when there are no free resources), this system call places the task in the queue of the specified semaphore, then changes it from the run state to the wait state (resource wait state).
The resource wait state is released upon the issue of a sig_sem, del_sem, or rel_wai system call, and the task returns to the ready state.

Remark When a task queues in the wait queue of the specified semaphore, it is executed in the sequence (FIFO sequence or priority order sequence) specified when that semaphore was generated (during configuration or when a cre_sem system call was issued).

Return value

*E_OK	0	Normal termination
E_ID	–35	Invalid ID number specification (maximum number of semaphores created < *semid*)
*E_NOEXS	–52	The specified semaphore does not exist.
E_OACV	–66	An unauthorized ID number (*semid* ≤ 0) was specified.
E_CTX	–69	Context error – The wai_sem system call was issued from a non-task. – The wai_sem system call was issued in the dispatch disabled state.
*E_DLT	–81	A specified semaphore was deleted by the del_sem system call.
*E_RLWAI	–86	The resource wait state was forcibly released by the rel_wai system call.
EV_SIGNAL	–225	The resource wait state was forcibly released by the vsnd_sig system call.

preq_sem

Pool and Request Semaphore (–107)

Task/nontask

Overview

Acquires resources (polling).

C format

```
#include      <stdrx.h>
ER            ercd = preq_sem(ID semid);
```

Parameter

I/O	Parameter	Description
I	ID *semid;*	Semaphore ID number

Explanation

This system call acquires resources from the semaphore specified in *semid* (the semaphore counter is decremented by 0x1).
When this system call is issued, if no resource can be acquired from a specified semaphore (when there are no free resources), this system returns E_TMOUT as the return value.

Return value

*E_OK	0	Normal termination
E_ID	–35	Invalid ID number specification (maximum number of semaphores created < *semid*)
*E_NOEXS	–52	The specified semaphore does not exist.
E_OACV	–66	An unauthorized ID number (*semid* ≤ 0) was specified.
*E_TMOUT	–85	The resource count for the specified semaphore is 0x0.

twai_sem

Wait on Semaphore with Timeout (–171)

Task

Overview

Acquires resources (with timeout).

C format

```
#include      <stdrx.h>
ER            ercd = twai_sem(ID semid, TMO, tmout);
```

Parameters

I/O	Parameter		Description	
I	ID	*semid;*	Semaphore ID number	
I	TMO	*tmout;*	Wait time	
			TMO_POL(0):	Quick return
			TMO_FEVR(1):	Permanent wait
			Value:	Wait time

Explanation

This system call acquires resources from the semaphore specified in *semid* (the semaphore counter is decremented by 0x1).
When this system call is issued, if no resource can be acquired from a specified semaphore (when there are no free resources), this system call places the task in the queue of the specified semaphore, then changes it from the run state to the wait state (resource wait state).
The resource wait state is released when the wait time specified in *tmout* elapses or when the sig_sem, del_sem, or rel_wai system call is issued, at which time it changes to the ready state.

Remark The task is queued into the queue of a specified semaphore in the order (FIFO order or priority order) specified when the semaphore was created (at configuration or upon the issue of cre_sem system call).

Return value

*E_OK	0	Normal termination
E_PAR	–33	Invalid wait time specification (*tmout* < TMO_FEVR)
E_ID	–35	Invalid ID number specification (maximum number of semaphores created < *semid*)
*E_NOEXS	–52	The specified semaphore does not exist.
E_OACV	–66	An invalid ID number (*semid* ≤ 0) was specified.
E_CTX	–69	Context error – The twai_sem system call was issued from a non-task. – The twai_sem system call was issued in the dispatch disabled state.
*E_DLT	–81	A specified semaphore was deleted by the del_sem system call.
*E_TMOUT	–85	Wait time elapsed.
*E_RLWAI	–86	The resource wait state was forcibly released by the issue of an rel_wai system call.
EV_SIGNAL	–225	The resource wait state was forcibly released by the vsnd_sig system call.

ref_sem

Refer Semaphore Status (–52)

Task/nontask

Overview

Acquires semaphore information.

C format

```
#include      <stdrx.h>
ER            ercd = ref_sem(T_RSEM *pk_rsem, ID semid);
```

Parameters

I/O	Parameter	Description
O	T_RSEM *pk_rsem;*	Start address of packet used to store semaphore information
I	ID *semid;*	Semaphore ID number

- Structure of semaphore information T_RSEM

```
typedef    struct     t_rsem {
           VP         exinf;      /*  Extended information        */
           BOOL_ID    wtsk;       /*  Existence of waiting task   */
           INT        semcnt;     /*  Current resource count      */
           INT        maxsem;     /*  Maximum resource count      */
} T_RSEM;
```

Explanation

This system call stores, into the packet specified in *pk_rsem*, the semaphore information (extended information, existence of waiting task, etc.) for the semaphore specified in *semid*.
Semaphore information is described in detail below.

exinf	...	Extended information
wtsk	...	Existence of waiting task FALSE(0): There is no waiting task Value: ID number of first task in queue
semcnt	...	Current resource count
maxsem	...	Maximum resource count specified at generation

Return value

*E_OK	0	Normal termination
E_PAR	–33	The start address of the packet used to store semaphore information is invalid (*pk_rsem* = 0).
E_ID	–35	Invalid ID number specification (maximum number of semaphores created < *semid*)
*E_NOEXS	–52	A specified semaphore does not exist.
E_OACV	–66	An unauthorized ID number (*semid* ≤ 0) was specified.

cre_flg

Create Event Flag (–41)

Task

Overview

Generates an event flag.

C format

- When an ID number is specified

```
#include        <stdrx.h>
ER              ercd = cve_flg(ID flgid, T_CFLG *pk_cflg);
```

- When an ID number is not specified

```
#include        <stdrx.h>
ER              ercd = cre_flg(ID_AUTO, T_CFLG *pk_cflg, ID *p_flgid);
```

Parameters

I/O	Parameter		Description
I	ID	*flgid*	Event flag ID number
I	T_CFLG	**pk_cflg*	Start address of packet storing event flag creation information
O	ID	**p_flgid*	Address of area used to store an ID number

- Structure of event flag creation information T_CFLG

```
 typedef   struct    t_cflg {
           VP        exinf;      /*  Extended information          */
           ATR       flgatr;     /*  Event flag attribute          */
           UINT      iflgptn;    /*  Initial bit pattern of event flag  */
} T_CFLG;
```

Explanation

RX4000 provides two types of interfaces for event flag creation: one in which an ID number must be specified and another in which an ID number is not specified.

- When an ID number is specified
  An event flag having the ID number specified in *flgid* is created based on the information specified in *pk_cflg*.

- When an ID number is not specified
  An event flag is created based on the information specified in *pk_cflg*.
  An ID number is allocated by RX4000 and the allocated ID number is stored into the area specified in *p_flgid*.

The following describes the event flag creation information in detail.

exinf	...	Extended information

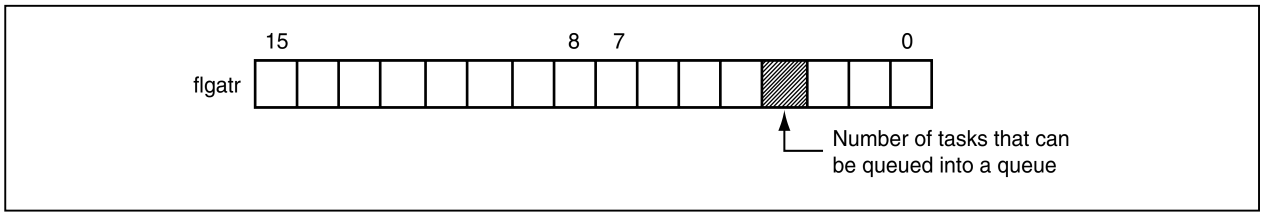
exinf is an area used for storing user-specific information on a specified event flag. The user can use this area as required.

Information set in exinf can be dynamically acquired by issuing the ref_flg system call from a processing program (task or non-task).

flgatr	...	Event flag attribute

Bit 3 .. Number of tasks that can be queued into a queue

TA_WSGL(0):	One task only
TA_WNUL(1):	Two or more tasks



iflgptn	...	Initial bit pattern of event flag

Return value

*E_OK	0	Normal termination
E_RSATR	–24	Invalid specification of attribute flgatr.
E_PAR	–33	Invalid parameter specification – The start address of the packet storing event flag creation information is invalid (*pk_cflg* = 0). – The address of the area used to store an ID number is invalid (*p_flgid* = 0). (When an event flag is created with no ID number specified)
E_ID	–35	Invalid ID number specification (maximum number of event flags created < *flgid*)
*E_OBJ	–63	An event flag having a specified ID number has already been created.
E_OACV	–66	An unauthorized ID number (*flgid* ≤ 0) was specified.
E_CTX	–69	The cre_flg system call was issued from a non-task.

del_flg

Delete Event Flag (–42)

Task

Overview

Deletes an event flag.

C format

```
#include      <stdrx.h>
ER            ercd = del_flg(ID flgid);
```

Parameter

I/O	Parameter	Description
I	ID *flgid*;	Event flag ID number

Explanation

This system call deletes the event flag specified in *flgid*.
The specified event flag is released from the control of RX4000.
The task released from the wait state (event flag wait state) by this system call has E_DLT returned as a return value of the system call (wai_flg or twai_flg) that initiated transition to the wait state.

Return value

*E_OK	0	Normal termination
E_ID	–35	Invalid ID number specification (maximum number of event flags created < *flgid*)
*E_NOEXS	–52	The specified event flag does not exist.
E_OACV	–66	An unauthorized ID number (*flgid* ≤ 0) was specified.
E_CTX	–69	The del_flg system call was issued from a non-task.

set_flg

Set Event Flag (–48)

Task/nontask

Overview

Sets a bit pattern.

C format

```
#include      <stdrx.h>
ER            ercd = set_flg(ID flgid, UINT setptn);
```

Parameters

I/O	Parameter	Description
I	ID *flgid;*	Event flag ID number
I	UINT *setptn;*	Bit pattern to be set (32-bit width)

Explanation

This system call executes logical OR between the bit pattern specified in *flgid* and that specified in *setptn*, and sets the result in a specified event flag.
For example, when this system call is issued, if the specified event flag's bit pattern is B'1100 and the bit pattern specified by *setptn* is B'1010, the bit pattern of the specified event flag becomes B'1110.
When this system call is issued, if the wait condition for a task queued in the queue of the specified event flag is satisfied, the task is removed from the queue.
Consequently, the relevant task changes from the wait state (event flag wait state) to the ready state, or from the wait_suspend state to the suspend state.

Return value

*E_OK	0	Normal termination
E_ID	–35	Invalid ID number specification (maximum number of event flags created < *flgid*)
*E_NOEXS	–52	A specified event flag does not exist
E_OACV	–66	An unauthorized ID number (*flgid* $\leq$ 0) was specified

clr_flg

Clear Event Flag (–47)

Task/nontask

Overview

Clears a bit pattern.

C format

```
#include        <stdrx.h>
ER              ercd = clr_flg(ID flgid, UINT clrptn);
```

Parameter

I/O	Parameter	Description
I	ID *flgid;*	Event flag ID number
I	UINT *clrptn;*	Bit pattern to clear (32-bit width)

Explanation

This system call executes logical AND between the bit pattern specified in *flgid* and that specified in *clrptn*, and sets the result in a specified event flag.

For example, when this system call is issued, if the specified event flag's bit pattern is B'1100 and the bit pattern specified in *cirptn* is B'1010, the specified event flag's bit pattern becomes B'1000.

Return value

*E_OK	0	Normal termination
E_ID	–35	Invalid ID number specification (maximum number of event flags created < *flgid*)
*E_NOEXS	–52	A specified event flag does not exist
E_OACV	–66	An unauthorized ID number (*flgid* ≤ 0) was specified

wai_flg

Wait Event Flag (–46)

Task

Overview

Checks a bit pattern.

C format

```
#include      <stdrx.h>
ER            ercd =  wai_flg(UINT *p_flgptn, ID flgid, UINT waiptn, UINT wfmode);
```

Parameters

I/O	Parameter		Description
O	UINT	**p_flgptn;*	Address of area used to store a bit pattern when a condition is satisfied
I	ID	*flgid;*	Event flag ID number
I	UINT	*waiptn;*	Request bit pattern (32-bit width)
I	UINT	*wfmode;*	Wait condition or condition satisfaction TWF_ANDW(0): AND wait TWF_ORW(2): OR wait TWF_CLR(1): Bit pattern is cleared

Explanation

This system call checks whether a bit pattern that satisfies the request bit pattern specified in *waiptn*, as well as the wait condition specified in *wfmode*, is set in the event flag specified in *flgid*.
If a bit pattern satisfying the wait condition is set in a specified event flag, this system call stores the bit pattern of the event flag in the area specified in *p_flgptn*.
When this system call is issued, if the bit pattern of the specified event flag does not satisfy the wait condition, this system call queues the task at the end of the queue for the specified event flag, then changes it from the run state to the wait state (event flag wait state).
The event flag wait state is released when a bit pattern satisfying the wait condition is set by the set_flg system call, or when the del_flg or rel_wai system call is issued, at which time it changes to the ready state.
The specification format for *wfmode* is shown below.

- *wfmode* = TWF_ANDW
  This system call checks whether all those bits of *waiptn* that are set to 1 are set in a specified event flag.

- *wfmode* = (TWF_ANDW|TWF_CLR)
  This system call checks whether all those bits of *waiptn* that are set to 1 are set in a specified event flag.
  If the wait condition is satisfied, the bit pattern for the specified event flag is cleared (B’0000 is set).

- *wfmode* = TWF_ORW
  This system call checks whether at least one of those bits of *waiptn* that are set to 1 is set in a specified event flag.

- *wfmode* = (TWF_ORW|TWF_CLR)
  This system call checks whether at least one of those bits of *waiptn* that are set to 1 is set in a specified event flag.
  If a wait condition is satisfied, the bit pattern of the specified event flag is cleared (B'0000 is set).

Cautions 1. RX4000 specifies the number of tasks that can be queued into the queue of an event flag, at generation (at configuration or upon the issue of a cre_flg system call).

TA_WSGL attribute: Only one task can be queued.
TA_WMUL attribute: Two or more tasks can be queued.

For this reason, if this system call is issued for the event flag having the TA_WSGL attribute for which waiting tasks are already queued, the wai_flg system call returns E_OBJ as the return value without performing bit pattern checking.

2. If the event flag wait state is forcibly released by issuing a del_flg or rel_wai system call, the contents of the area specified in *p_flgptn* will be undefined.

Return value

*E_OK	0	Normal termination
E_PAR	–33	Invalid parameter specification – The address of the area used to store a bit pattern when a condition is satisfied is invalid (*p_flgptn* = 0). – A request bit pattern is incorrectly specified (*waiptn* = 0). – The wait condition or condition satisfaction parameter *wfmode* is incorrectly specified.
E_ID	–35	Invalid ID number specification (maximum number of event flags created < *flgid*)
*E_NOEXS	–52	A specified event flag does not exist.
*E_OBJ	–63	The wai_flg system call was issued for the event flag having the TA_WSGL attribute in which waiting tasks were already queued.
E_OACV	–66	An unauthorized ID number (*flgid* ≤ 0) was specified.
E_CTX	–69	Context error – The wai_flg system call was issued from a non-task. – The wai_flg system call was issued from the dispatch disabled state.
*E_DLT	–81	A specified event flag was deleted by a del_flg system call.
*E_RLWAI	–86	The event flag wait state was forcibly released by an rel_wai system call.
EV_SIGNAL	–225	The event flag wait state was forcibly released by the vsnd_sig system call.

pol_flg

Poll Event Flag (–106)

Task/nontask

Overview

Checks a bit pattern (polling).

C format

```
#include      <stdrx.h>
ER            ercd =  pol_flg(UINT *p_flgptn, ID flgid, UINT waiptn, UINT wfmode);
```

Parameters

I/O	Parameter		Description
O	UINT	**p_flgptn;*	Address of area used to store a bit pattern when a condition is satisfied
I	ID	*flgid;*	Event flag ID number
I	UINT	*waiptn;*	Request bit pattern (32-bit width)
I	UINT	*wfmode;*	Wait condition or condition satisfaction TWF_ANDW(0): AND wait TWF_ORW(2): OR wait TWF_CLR(1): Bit pattern is cleared.

Explanation

This system call checks whether a bit pattern satisfying both the request bit pattern specified in *waiptn* and the wait condition specified in *wfmode* is set in the event flag specified in *flgid*.
If a bit pattern satisfying the wait condition is set in a specified event flag, this system call stores the bit pattern of the event flag into the area specified in *p_flgptn*.
When this system call is issued, if the bit pattern of a specified event flag does not satisfy the wait condition, this system call returns E_TMOUT as the return value.
The *wfmode* specification format is shown below.

- *wfmode* = TWF_ANDW
  This system call checks whether all those bits of *waiptn* that are set to 1 are set in a specified event flag.

- *wfmode* = (TWF_ANDW|TWF_CLR)
  This system call checks whether all those bits of *waiptn* that are set to 1 are set in a specified event flag.
  If the wait condition is satisfied, the bit pattern for the specified event flag is cleared (B’0000 is set).

- *wfmode* = TWF_ORW
  This system call checks whether at least one of those bits of *waiptn* that are set to 1 is set in a specified event flag.

- *wfmode* = (TWF_ORW|TWF_CLR)
  This system call checks whether at least one of those bits of *waiptn* that are set to 1 is set in a specified event flag.
  If the wait condition is satisfied, the bit pattern for the specified event flag is cleared (B'0000 is set).

Caution RX4000 specifies the number of tasks that can be queued into the queue of an event flag, at generation (at configuration or upon the issue of a cre_flg system call).

TA_WSGL attribute: Only one task can be queued.
TA_WMUL attribute: Two or more tasks can be queued.

For this reason, if this system call is issued for an event flag having the TA_WSGL attribute in which waiting tasks are already queued, the wai_flg system call returns E_OBJ as the return value without performing bit pattern checking.

Return value

*E_OK	0	Normal termination
E_PAR	–33	Invalid parameter specification – The address of the area used to store a bit pattern when a condition is satisfied is invalid (*p_flgptn* = 0). – A request bit pattern is incorrectly specified (*waiptn* = 0). – The wait condition or condition satisfaction parameter *wfmode* is incorrectly specified.
E_ID	–35	Invalid ID number specification (maximum number of event flags created < *flgid*)
*E_NOEXS	–52	A specified event flag does not exist.
*E_OBJ	–63	This pol_flg system call was issued for the event flag of TA_WSGL attribute in which waiting tasks are already queued.
E_OACV	–66	An unauthorized ID number (*flgid* ≤ 0) was specified.
*E_TMOUT	–85	The bit pattern of the specified event flag does not satisfy the wait condition.

twai_flg

Wait Event Flag with Timeout (–170)

Task

Overview

Checks a bit pattern (with timeout).

C format

```
#include      <stdrx.h>
ER            ercd =  twai_flg(UINT *p_flgptn, ID flgid, UINT waiptn, UINT wfmode,
              TMO tmout);
```

Parameters

I/O	Parameter		Description	
O	UINT	**p_flgptn;*	Address of area used to store a bit pattern when a condition is satisfied	
I	ID	*flgid;*	Event flag ID number	
I	UINT	*waiptn;*	Request bit pattern (32-bit width)	
I	UINT	*wfmode;*	Wait condition or condition satisfaction	
			TWF_ANDW(0):	AND wait
			TWF_ORW(2):	OR wait
			TWF_CLR(1):	Bit pattern is cleared.
I	TMO	*tmount;*	Wait time (basic clock cycles)	
			TMO_POL(0):	Quick return
			TMO_FEVR(–1):	Permanent wait
			Value:	Wait time

Explanation

This system call checks whether a bit pattern satisfying both the request bit pattern specified in *waiptn* and the wait condition specified in *wfmode* is set in the event flag specified in *flgid*.

If a bit pattern satisfying wait condition is set in a specified event flag, this system call stores the bit pattern of the event flag into the area specified in *p_flgptn*.

Upon the issue of this system call, if the bit pattern of the specified event flag does not satisfy the wait condition, this system call queues the task at the end of the queue for a specified event flag, then changes it from the run state to the wait state (event flag wait state).

The event flag wait state is released upon the elapse of the wait time specified in *tmout*, when a bit pattern satisfying wait condition is set by the set_flg system call, or when the del_flg or rel_wai system call is issued, at which time the task returns to the ready state.

The *wfmode* specification format is shown below.

- *wfmode* = TWF_ANDW
  This system call checks whether all those bits of *waiptn* that are set to 1 are set in a specified event flag.

- *wfmode* = (TWF_ANDW|TWF_CLR)
  This system call checks whether all those bits of *waiptn* that are set to 1 are set in a specified event flag.
  If the wait condition is satisfied, the bit pattern for the specified event flag is cleared (B'0000 is set).

- *wfmode* = TWF_ORW
  This system call checks whether at least one of those bits of *waiptn* that are set to 1 is set in a specified event flag.

- *wfmode* = (TWF_ORW|TWF_CLR)
  This system call checks whether at least one of those bits of *waiptn* that are set to 1 is set in a specified event flag.
  If the wait condition is satisfied, the bit pattern of the specified event flag is cleared (B'0000 is set).

Cautions 1. RX4000 specifies the number of tasks that can be queued into the queue of the event flag, at generation (at configuration or upon the issue of a cre_flg system call).

TA_WSGL attribute: Only one task can be queued.
TA_WMUL attribute: Two or more tasks can be queued.

For this reason, if this system call is issued for an event flag having the TA_WSGL attribute in which waiting tasks are already queued, this system call returns E_OBJ as the return value without performing bit pattern checking.

2. If the event flag wait state is forcibly released by a del_flg or rel_wai system call, the contents of the area specified in *p_flgptn* will be undefined.

Return value

*E_OK	0	Normal termination
E_PAR	–33	Invalid parameter specification – The address of the area used to store a bit pattern when a condition is satisfied is invalid (*p_flgptn* = 0). – The specification of a request bit pattern is invalid (*waiptn* = 0). – The specification of a wait condition or condition satisfaction parameter *wfmode* is invalid. – Invalid wait time specification (*tmout* < TMO_FEVR)
E_ID	–35	Invalid ID number specification (maximum number of event flags created < *flgid*)
*E_NOEXS	–52	A specified event flag does not exist.
*E_OBJ	–63	This twai_flg system call was issued for the event flag having the TA_WSGL attribute in which waiting tasks were already queued.
E_OACV	–66	An unauthorized ID number (*flgid* ≤ 0) was specified.
E_CTX	–69	Context error – The twai_flg system call was issued from a non-task. – The twai_flg system call was issued from the dispatch disabled state.
*E_DLT	–81	A specified event flag was deleted by the issue of a del_flg system call.
*E_TMOUT	–85	Wait time elapsed.
*E_RLWAI	–86	The event flag wait state was forcibly released by the issue of an rel_wai system call.
EV_SIGNAL	–225	The event flag wait state was forcibly released by the issue of an vsnd_sig system call.

Refer Event Flag Status (–44)

ref_flg

Task/nontask

Overview

Acquires event flag information.

C format

```
#include      <stdrx.h>
ER            ercd = ref_flg(T_RFLG *pk_rflg, ID flgid);
```

Parameters

I/O	Parameter	Description
O	T_RFLG *pk_rflg;	Start address of packet used to store event flag information
I	ID flgid;	Event flag ID number

- Structure of event flag information T_RFLG

```
typedef    struct     t_rflg {
           VP         exinf;        /*  Extended information        */
           BOOL_ID    wtsk;         /*  Existence of waiting task   */
           UINT       flgptn;       /*  Current bit pattern         */
} T_RFLG;
```

Explanation

This system call stores, in the packet specified in *pk_rflg*, the event flag information (extended information, existence of waiting task, etc.) for the event flag specified in *flgid*.
Event flag information is described in detail below.

exinf ... Extended information

wtsk ... Existence of waiting task

FALSE(0):	There is no waiting task.
Value:	ID number of first task in queue

flgptn ... Current bit pattern

Return value

*E_OK	0	Normal termination
E_PAR	–33	The start address of the packet used to store event flag information is invalid (*pk_flg* = 0).
E_ID	–35	Invalid ID number specification (maximum number of event flags created < *flgid*)
*E_NOEXS	–52	A specified event flag does not exist.
E_OACV	–66	An unauthorized ID number (*flgid* ≤ 0) was specified.

cre_mbx

Create Mailbox (–57)

Task

Overview

Generates a mailbox.

C format

- When an ID number is specified

```
#include        <stdrx.h>
ER              ercd = cre_mbx(ID mbxid, T_CMBX *pk_cmbx);
```

- When an ID number is not specified

```
#include             <stdrx.h>
ER                   ercd = cre_mbx(ID_AUTO, T_CMBX *pk_cmbx, ID *p_mbxid);
```

Parameters

I/O	Parameter	Description
I	ID *mbxid;*	Mailbox ID number
I	T_CMBX **pk_cmbx;*	Start address of packet used to store mailbox creation information
O	ID **p_mbxid;*	Address of area used to store an ID number

- Structure of mailbox creation information T_CMBX

```
typedef    struct    t_cmbx {
           VP        exinf;      /*  Extended information       */
           ATR       mbxatr;     /*  Mailbox attribute          */
} T_CMBX;
```

Explanation

RX4000 provides two types of interfaces for mailbox creation: one in which an ID number must be specified for mailbox creation, and another in which an ID number is not specified.

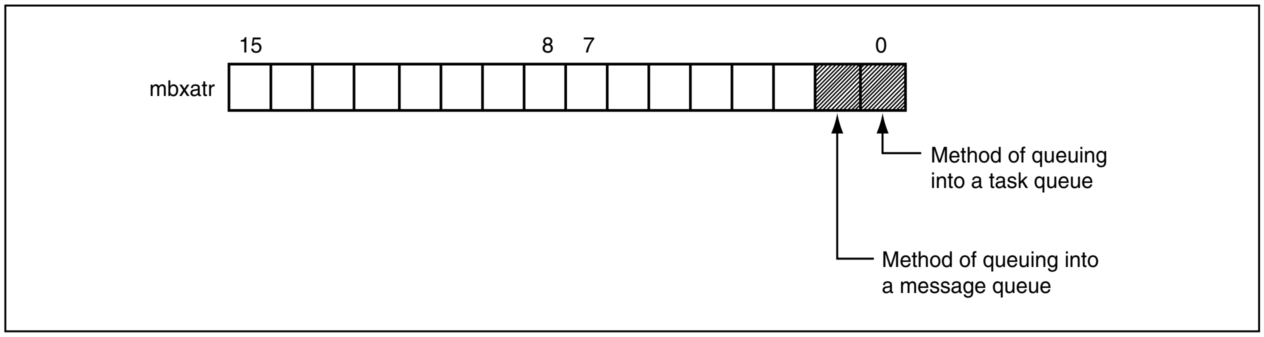
- When an ID number is specified
  An mailbox having an ID number specified in *mbxid* is created based on the information specified in *pk_cmbx*.

- When an ID number is not specified
  An mailbox is created based on the information specified in *pk_cmbx*.
  An ID number is allocated by RX4000. The allocated ID number is stored into the area specified in *p_mbxid*.

The following describes the mailbox creation information in detail.

exinf	...	Extended information exinf is an area used for storing user-specific information on a specified mailbox. The user can use this area as required. Information set in exinf can be dynamically acquired by issuing the ref_mbx system call from a processing program (task/non-task).
mbxatr	...	Mailbox attribute

Bit 0	..	Method of queuing into a task queue	
		TA_TPRI(0):	Priority order
		TA_TFIFO(1):	FIFO order
Bit 1	..	Method of queuing into a message queue	
		TA_MPRI(0):	Priority order
		TA_MFIFO(1):	FIFO order


15
8
7
0
mbxatr
Method of queuing into a task queue
Method of queuing into a message queue

Return value

*E_OK	0	Normal termination
E_RSATR	–24	Invalid specification of attribute mbxatr
E_PAR	–33	Invalid parameter specification – The start address of the packet storing the mailbox creation information is invalid (*pk_cmbx* = 0). – The address of the area used to store an ID number is invalid (*p_mbxid* = 0). (When a mailbox is created without an ID number specified)
E_ID	–35	Invalid ID number specification (maximum number of mailboxes created < *mbxid*)
*E_OBJ	–63	A mailbox having the specified ID number has already been created.
E_OACV	–66	An unauthorized ID number (*mbxid* ≤ 0) was specified.
E_CTX	–69	The cre_mbx system call was issued from a non-task.

del_mbx

Delete Mailbox (–58)

Task

Overview

Deletes a mailbox.

C format

```
#include      <stdrx.h>
ER            ercd = del_mbx(ID mbxid);
```

Parameters

I/O	Parameter	Description
I	ID *mbxid;*	Mailbox ID number

Explanation

This system call deletes the mailbox specified in *mbxid*.
The specified mailbox is released from the control of RX4000.
The task released from the wait state (message wait state) by this system call has E_DLT returned as the return value of the system call (rcv_msg or trcv_msg) that instigated transition to the wait state.

Remark When this system call is issued, any message using a memory block acquired from a memory pool is queued into the message queue of a specified mailbox, then the message is returned to the memory pool.

Return value

*E_OK	0	Normal termination
E_ID	–35	Invalid ID number specification (maximum number of mailboxes created < *mbxid*)
*E_NOEXS	–52	The specified mailbox does not exist.
E_OACV	–66	An unauthorized ID number (*mbxid* ≤ 0) was specified.
E_CTX	–69	The del_mbx system call was issued from a non-task.

snd_msg

Send Message (–63)

Task/nontask

Overview

Sends a message.

C format

```
#include      <stdrx.h>
ER            ercd = snd_msg(ID mbxid, T_MSG *pk_msg);
```

Parameters

I/O	Parameter	Description
I	ID *mbxid;*	Mailbox ID number
I	T_MSG **pk_msg;*	Start address of packet used to store a message

- Structure of message T_MSG

```
typedef   struct    t_msg {
          VW        msgrfu;      /*  Message management area   */
          PRI       msgpri;      /*  Message priority          */
          VB        msgcont[];   /*  Message body              */
} T_MSG;
```

Explanation

This system call sends the message specified in *pk_msg* to the mailbox specified in *mbxid* (queues the message into a message queue).
When this system call is issued, if a task is queued into the task queue of a specified mailbox, this system call passes the message to the task (first task in the task queue) without performing message queuing.
Consequently, the relevant task is removed from the task queue, and its state changes from the wait state (message wait state) to the ready state, or from the wait_suspend state to the suspend state.

Remark When a message queues in the message wait queue of the specified mailbox, it is executed in the sequence (FIFO sequence or priority order sequence) specified when that mailbox was generated (during configuration or when a cre_mbx system call was issued).

Caution **RX4000 uses the first 4 bytes (message management area msgrfu) of a message as a link area for enabling queuing into a message queue. Accordingly, sending a message to a specified mailbox requires that 0x0 be set in msgrfu before issuing the snd_msg system call.**
If a value other than 0x0 is set in msgrfu when the snd_msg system call is issued, RX4000 recognizes that the relevant message is already queued into a message queue, and this system call returns E_OBJ as a return value without sending the message.

Return value

*E_OK	0	Normal termination
E_PAR	–33	The start address of a packet used to store a message is invalid (*pk_msg* = 0).
E_ID	–35	Invalid ID number specification (maximum number of mailboxes created < *mbxid*)
*E_NOEXS	–52	A specified mailbox does not exist.
E_OBJ	–63	The area specified for a message is already being used for messages.
E_OACV	–66	An unauthorized ID number (*mbxid* ≤ 0) was specified.

rcv_msg

Receive Message from Mailbox (–61)

Task

Overview

Receives a message.

C format

```
#include        <stdrx.h>
ER              ercd = rcv_msg(T_MSG **ppk_msg, ID mbxid);
```

Parameters

I/O	Parameter		Description
O	T_MSG	***ppk_msg;*	Address of area used to store the start address of a message
I	ID	*mbxid;*	Mailbox ID number

Explanation

This system call receives a message from the mailbox specified in *mbxid* and stores its start address into the area specified in *ppk_msg*.
When this system call is issued, if a message cannot be received from a specified mailbox (when no message exists in a message queue), this system call queues the task into the task queue of the specified mailbox, then changes its state from the run state to the wait state (message wait state).
The message wait state is released when the snd_msg, del_mbx, or rel_wai system call is issued, and the task returns to the ready state.

Remark When a task queues in the task wait queue of the specified mailbox, it is executed in the sequence (FIFO sequence or priority order sequence) specified when that mailbox was generated (during configuration or when a cre_sem system call was issued).

Return value

*E_OK	0	Normal termination
E_PAR	–33	The address of the area used to store the start address of a message is invalid (*ppk_msg* = 0).
E_ID	–35	Invalid ID number specification (maximum number of mailboxes created < *mbxid*)
*E_NOEXS	–52	A specified mailbox does not exist.
E_OACV	–66	An unauthorized ID number (*mbxid* ≤ 0) was specified.
E_CTX	–69	Context error – The rcv_msg system call was issued from a non-task. – The rcv_msg system call was issued from the dispatch disabled state.
*E_DLT	–81	A specified mailbox was deleted by a del_mbx system call.
*E_RLWAI	–86	The message wait state was forcibly released by an rel_wai system call.
EV_SIGNAL	–225	The message wait state was forcibly released by the vsnd_sig system call.

prcv_msg

Poll and Receive Message from Mailbox (–108)

Task/nontask

Overview

Receives a message (polling).

C format

```
#include      <stdrx.h>
ER            ercd = prcv_msg(T_MSG **ppk_msg, ID mbxid);
```

Parameters

I/O	Parameter	Description
O	T_MSG ***ppk_msg;*	Address of area used to store the start address of a message
I	ID *mbxid;*	Mailbox ID number

Explanation

This system call receives a message from the mailbox specified in *mbxid* and stores its start address into the area specified in *ppk_msg*.

When this system call is issued, if a message cannot be received from a specified mailbox (when no message exists in the message queue), E_TMOUT is returned as the return value.

Return value

*E_OK	0	Normal termination
E_PAR	–33	The address of the area to store the start address of a message is invalid (*ppk_msg* = 0).
E_ID	–35	Invalid ID number specification (maximum number of mailboxes created < *mbxid*)
*E_NOEXS	–52	A specified mailbox does not exist.
E_OACV	–66	An unauthorized ID number (*mbxid* ≤ 0) was specified.
*E_TMOUT	–85	No message exists in a specified mailbox.

Receive Message from Mailbox with Timeout (–172)

trcv_msg

Task

Overview

Receives a message (with timeout).

C format

```
#include      <stdrx.h>
ER            ercd = trcv_msg(T_MSG **ppk_msg, ID mbxid, TMO tmout);
```

Parameters

I/O	Parameter		Description	
O	T_MSG	***ppk_msg;*	Address of area used to store the start address of a message	
I	ID	*mbxid;*	Mailbox ID number	
I	TMO	*tmout;*	Wait time (basic clock cycles)	
			TMO_POL(0):	Quick return
			TMO_FEVR(–1):	Permanent wait
			Value:	Wait time

Explanation

This system call receives a message from the mailbox specified in *mbxid* and stores its start address into the area specified in *ppk_msg*.

When this system call is issued, if a message cannot be received from a specified mailbox (when no message exists in the message queue), this system call queues the task into the task queue of the specified mailbox, then changes its state from the run state to the wait state (message wait state).

The message wait state is released when the wait time specified in *tmout* elapses or when the snd_msg, del_mbx, or rel_wai system call is issued, and the task returns to the ready state.

Remark When a task queues in the task wait queue of the specified mailbox, it is executed in the sequence (FIFO sequence or priority order sequence) specified when that mailbox was generated (during configuration or when a cre_mbx system call was issued).

Return value

*E_OK	0	Normal termination
E_PAR	–33	Invalid parameter specification – The address of the area used to store the start address of a message is invalid (*ppk_msg* = 0). – Invalid wait time specification (*tmout* < TMO_FEVR)
E_ID	–35	Invalid ID number specification (maximum number of mailboxes created < *mbxid*)
*E_NOEXS	–52	A specified mailbox does not exist.
E_OACV	–66	An unauthorized ID number (*mbxid* ≤ 0) was specified.
E_CTX	–69	Context error – The trcv_msg system call was issued from a non-task. – The trcv_msg system call was issued from the dispatch disabled state.
*E_DLT	–81	A specified mailbox was deleted by a del_mbx system call.
*E_TMOUT	–85	Wait time elapsed.
*E_RLWAI	–86	The message wait state was forcibly released by an rel_wai system call.
EV_SIGNAL	–225	The message wait state was forcibly released by an vsnd_sig system call.

ref_mbx

Refer Mailbox Status (–60)

Task/nontask

Overview

Acquires mailbox information.

C format

```
#include      <stdrx.h>
ER            ercd = ref_mbx(T_RMBX *pk_rmbx, ID mbxid);
```

Parameters

I/O	Parameter	Description
O	T_RMBX *pk_rmbx;	Start address of packet used to store mailbox information
I	ID mbxid;	Mailbox ID number

- Structure of mailbox information T_RMBX

```
typedef    struct     t_rmbx {
           VP         exinf;       /*  Extended information          */
           BOOL_ID    wtsk;        /*  Existence of waiting task     */
           T_MSG      *pk_msg;     /*  Existence of waiting message  */
} T_RMBX;
```

Explanation

This system call stores mailbox information (extended information, existence of waiting task, etc.) for the mailbox specified in *mbxid* into the packet specified in *pk_rmbx*.
Mailbox information is described in detail below.

exinf ... Extended information

wtsk ... Existence of waiting task

FALSE(0):	No waiting task
Value:	ID number of the first message of queue

pk_msg ... Existence of waiting message

NADR(–1):	No waiting message
Value:	Address of the first message of queue

Return value

*E_OK	0	Normal termination
E_PAR	–33	The start address of the packet to store mailbox information is invalid (*pk_rmbx* = 0).
E_ID	–35	Invalid ID number specification (maximum number of mailboxes created < *mbxid*)
*E_NOEXS	–52	A specified mailbox does not exist.
E_OACV	–66	An unauthorized ID number (*mbxid* ≤ 0) was specified.

12.8.5 Interrupt management system calls

This section explains a group of system calls (interrupt management system calls) that perform processing that depends on maskable interrupts.

Table 12-9 lists the interrupt management system calls.

Table 12-9. Interrupt Management System Calls

System call	Function
def_int	Registers an interrupt handler and cancels its registration.
ret_int	Returns from an interrupt handler.
ret_wup	Wakes up another task and returns from an interrupt handler.
loc_cpu	Disables the acceptance of maskable interrupts and dispatch processing.
unl_cpu	Enables the acceptance of maskable interrupts and dispatch processing.
dis_int	Disables acceptance of maskable interrupts.
ena_int	Enables acceptance of maskable interrupts.
chg_ims	Changes the interrupt mask.
ref_ims	Acquires the interrupt mask.

def_int

Define Interrupt Handler (–65)

Task/nontask

Overview

Registers an interrupt handler and cancels its registration.

C format

```
#include      <stdrx.h>
ER            ercd = def_int(UINT dintno, T_DINT *pk_dint);
```

Parameters

I/O	Parameter	Description
I	UINT *dintno*;	Interrupt level of interrupt handler
I	T_DINT **pk_dint*;	Start address of packet storing interrupt handler registration information

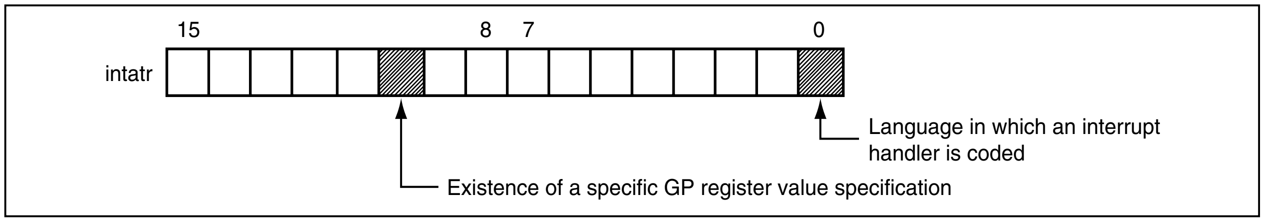
- Structure of interrupt handler registration information T_DINT

```
typedef    struct    t_dint {
           ATR       intatr;    /* Attribute of interrupt handler                  */
           FP        inthdr;    /* Activation address of interrupt handler         */
           VP        gp;        /* Specific GP register value for interrupt handler */
} T_DINT;
```

Explanation

This system call uses the information specified in *pk_dint* to register the indirectly activated interrupt handler activated upon the occurrence of a maskable interrupt of the interrupt level specified in *dintno*.
Indirectly activated interrupt handler registration information is described in detail below.

intatr ... Attribute of interrupt handler
  Bit 0 .. Language in which an interrupt handler is coded
    TA_ASM(0): Assembly language
    TA_HLNG(1): C
  Bit 10 .. Existence of a specific GP register value specification
    TA_DPID(1): Specifies a specific GP register value.


15
8
7
0
intatr
Language in which an interrupt handler is coded
Existence of a specific GP register value specification

inthdr ... Activation address of interrupt handler
gp ... Specific GP register value for interrupt handler

When this system call is issued, if an interrupt handler corresponding to a specified interrupt level has already been registered, this system call does not handle this as an error and newly registers the specified interrupt handler.

When this system call is issued, if NADR(–1) is set in the area specified in *pk_dint*, the registration of the interrupt handler specified in *dintno* is canceled.

Remark When the value 1 of bit 10 of intatr is other than TA_DPID, the contents of gp are meaningless.

Return value

*E_OK	0	Normal termination
E_RSATR	–24	Invalid specification of attribute intatr
E_PAR	–33	Invalid parameter specification – Invalid interrupt level specification (8 ≤ *dintno*) – The start address of the packet storing interrupt handler registration information is invalid (*pk_dint* = 0). – Invalid specification of the activation address (inthdr = 0)

ret_int

Return from Interrupt Handler (–69)

Interrupt handler

Overview

Returns from a interrupt handler.

C format

```
#include      <stdrx.h>
ret_int();
```

Parameter

None.

Explanation

This system call returns from a interrupt handler.
If a system call (chg_pri, sig_sem, etc.) requiring task scheduling is issued from a interrupt handler, RX4000 merely queues the tasks into the queue and delays actual scheduling until a system call (issuing ret_int or ret_wup system call) is issued to return from the directly activated interrupt handler. Then, the queued tasks are all performed at one time.
Furthermore, this system call function is offered as a macro (return 0) in RX4000.

Cautions This system call does not notify the external interrupt controller of the termination of processing (issue of EOI command). Accordingly, for return from the interrupt handler activated by an external interrupt request, the external interrupt controller must be notified of termination before the issue of this system call.

Return value

None.

ret_wup

Return and Wakeup Task (–70)

Interrupt handler

Overview

Wakes up another task and returns from a interrupt handler.

C format

```
#include      <stdrx.h>
ret_wup(ID tskid);
```

Parameter

I/O	Parameter	Description
I	ID *tskid;*	Task ID number

Explanation

This system call returns from a interrupt handler after the issue of a wake-up request to the task specified in *tskid* (the wake-up request counter is incremented by 0x1).
When this system call is issued, if the specified task is in the wait state (wake-up wait state), without issuing a wake-up request (incrementing the wake-up request counter), this system call changes the specified task from the wake-up wait state to the ready state.
If a system call (chg_pri, sig_sem, etc.) requiring task scheduling is issued from a interrupt handler, RX4000 merely queues the tasks into a queue and delays the actual scheduling until a system call (issuing ret_int or ret_wup system call) is issued to return from the directly activated interrupt handler. Then, the queued tasks are all performed at one time.
This system call function is offered as a macro (return (tskid)) in RX4000.

Cautions 1. This system call does not notify the external interrupt controller of processing termination (issue of the EOI command). Accordingly, for return from the directly activated interrupt handler activated by an external interrupt request, the external interrupt controller must be notified of processing termination before the issue of this system macro.

2. In this system call, if the following types of error occur, only processing for returning from the interrupt handler is performed.

- Invalid ID number specification (maximum number of tasks created < *tskid*).
- A specified task does not exist.
- A specified task is in the run or dormant state.
- An unauthorized ID number (*tskid* ≤ 0) was specified.
- The number of wake-up requests exceeded 127.

Return value

None.

loc_cpu

Lock CPU (–8)

Task

Overview

Disables the acceptance of maskable interrupts and dispatch processing.

C format

```
#include      <stdrx.h>
ER            ercd = loc_cpu(void);
```

Parameter

None.

Explanation

This system call disables the acceptance of maskable interrupts and dispatch processing (task scheduling).
Actually, this system call clears the interrupt enable (IE) bit from the processor's status register and disables acceptance of all maskable interrupts.
Therefore, for the period of time from issue of this system call to issue of the unl_cpu, there is no transfer of control to another handler or task.
If a maskable interrupt occurs after this system call is issued but before the unl_cpu system call is issued, RX4000 delays processing for the interrupt (interrupt handler) until the unl_cpu system call is issued. If a system call (chg_pri, sig_sem, etc) requiring task scheduling is issued, RX4000 merely queues the tasks into a queue and delays the actual scheduling until the unl_cpu system call is issued. Then, all the tasks are performed at one time.

Caution This system call does not queue disable requests. Accordingly, if this system call has already been and the acceptance of maskable interrupts and dispatch processing has been disabled, the system does not handle this as an error and performs no processing.

Return value

*E_OK	0	Normal termination
E_CTX	–69	The loc_cpu system call was issued from a non-task.

unl_cpu

Unlock CPU (–7)

Task

Overview

Enables the acceptance of maskable interrupts and dispatch processing.

C format

```
#include      <stdrx.h>
ER            ercd = unl_cpu(void);
```

Parameter

None.

Explanation

This system call resumes the acceptance of maskable interrupts and dispatch processing (task scheduling) disabled by the loc_cpu system call.
Actually, this system call sets the interrupt enable (IE) bit from the processor's status register and enables acceptance of all maskable interrupts.
If a maskable interrupt occurs after the loc_cpu system call is issued but before this system call is issued, RX4000 delays processing for the interrupt (interrupt handler) until this system call is issued. If a system call (chg_pri, sig_sem, etc) requiring task scheduling is issued, RX4000 merely queues the tasks into a queue and delays actual scheduling until the unl_cpu system call is issued. Then all the tasks are performed at one time.

Remark Dispatch processing that was disabled by the issue of the dis_dsp system call is resumed by this system call.

Caution **This system call does not queue resume requests. Accordingly, if this system call has already been issued, maskable interrupts have been accepted, and dispatch processing has been resumed, this system call does not handle this as an error and performs no processing.**

Return value

*E_OK	0	Normal termination
E_CTX	–69	The unl_cpu system call was issued from a non-task.

dis_int

Disable Interrupt (–72)

Task/nontask

Overview

Disables acceptance of maskable interrupts.

C format

```
#include      <stdrx.h>
ER            ercd = dis_int(void);
```

Parameter

None.

Explanation

This disables acceptance of maskable interrupts. Actually, this system call clears the interrupt enable (IE) bit from the processor's status register and disables acceptance of all maskable interrupts.

Cautions 1. With this system call, no disable request queuing is performed. Therefore, if this system call is issued already and reception of maskable interrupts is disabled, non processing is executed, it is not treated as an error.

2. In RX4000, a software timer is configured using a timer interrupt. Therefore, if interrupts are disabled, operations such as delayed wake-up, time out cyclic activation no longer function.

Return value

*E_OK	0	Normal termination
E_CTX	–69	The dis_int system call was issued from the interrupt disable state and the dispatch disable state.

ena_int

Enable Interrupt (–71)

Task/nontask

Overview

Enables acceptance of a maskable interrupt.

C format

```
#include      <stdrx.h>
ER            ercd = ena_int(void);
```

Parameter

None.

Explanation

Sets the interrupt enable (IE) bit in the processor's status register and enables acceptance of all maskable interrupts.

Cautions 1. With this system call, reopen request queuing is not performed. Therefore, if this system call has been issued already, if acceptance of maskable interrupts has been enabled, no processing is executed and it is not treated as an error.

2. With this system call, the interrupt mask (IM) bit in the status register is not changed. Therefore, if there are any cleared interrupt mask (IM) bits in the status register when this system call is issued, the affected interrupt is not received. When setting the interrupt mask (IM) bit in the status register, use the chg_ims system call.

Return value

*E_OK	0	Normal termination
E_CTX	–69	The ena_int system call was issued from the interrupt disabled state and the dispatch disabled state.

chg_ims

Change Interrupt Mask (–67)

Task/nontask

Overview

Changes the interrupt mask.

C format

```
#include      <stdrx.h>
ER            ercd = chg_ims(UINT intms);
```

Parameter

I/O	Parameter	Description
I	UNIT *intms;*	Interrupt mask

Explanation

This system call changes the interrupt mask of the processor to the value specified in *intms*.
Actually, this system call rewrites the value in the status register's interrupt mask (IM) area to the value in *intms*.

Cautions 1. Bit 7 of the IM area corresponds to the timer interrupt. In RX4000, since the software timer is configured using timer interrupts, if this is disabled, delayed wake-up, time out, cyclic activation, etc. stop functioning.

2. This system call does not change the interrupt enable (IE) bit in the status register. Therefore, when the interrupt enable (IE) bit in the status register is cleared, even if this system call is issued, interrupts are not received. Issue the unl_cpu system call to set the status register's interrupt enable (IE) bit.

Return value

*E_OK	0	Normal termination
E_PAR	–33	Invalid specification of interrupt mask (0xFF< *intms*)

ref_ims

Refer Interrupt Mask (–68)

Task/nontask

Overview

Acquires the interrupt mask.

C format

```
#include      <stdrx.h>
ER            ercd = ref_ims(UINT *p_intms);
```

Parameter

I/O	Parameter	Description
O	UINT *p_intms;	Address of area used to store interrupt mask

Explanation

This system call stores the interrupt mask of the processor in the area specified in *p_intms*.

Return value

*E_OK	0	Normal termination
E_PAR	–33	The address of the area used to store interrupt mask is invalid (*p_intms* = 0).

12.8.6 Memory pool management system calls

This section explains a group of system calls that allocate memory blocks (memory pool management system calls).

Table 12-10 lists the memory pool management system calls.

Table 12-10. Memory Pool Management System Calls

System call	Function
cre_mpl	Generates a memory pool.
del_mpl	Deletes a memory pool.
get_blk	Acquires a memory block.
pget_blk	Acquires a memory block (polling).
tget_blk	Acquires a memory block (with timeout).
rel_blk	Returns a memory block.
ref_mpl	Acquires memory pool information.

cre_mpl

Create Variable-size Memory Pool (–137)

Task

Overview

Generates a memory pool.

C format

- When an ID number is specified

```
#include        <stdrx.h>
ER              ercd = cre_mpl(ID mplid, T_CMPL *pk_cmpl);
```

- When an ID number is not specified

```
#include        <stdrx.h>
ER              ercd = cre_mpl(ID_AUTO, T_CMPL *pk_cmpl, ID *p_mplid);
```

Parameters

I/O	Parameter	Description
I	ID *mplid*;	Memory pool ID number
I	T_CMPL **pk_cmpl*;	Start address of packet containing memory pool creation information
O	ID **p_mplid*;	Address of area used to store an ID number

- Structure of memory pool creation information T_CMPL

```
typedef   struct   t_cmpl {
          VP       exinf;     /*  Extended information      */
          ATR      mplatr;    /*  Memory pool attribute     */
          INT      mplsz;     /*  Memory pool size          */
} T_CMPL;
```

Explanation

RX4000 provides two types of interfaces for memory pool creation: one in which an ID number must be specified for memory pool creation, and another in which an ID number is not specified.

- When an ID number is specified
  A memory pool having an ID number specified in *mplid* is created based on the information specified in *pk_cmpl*.

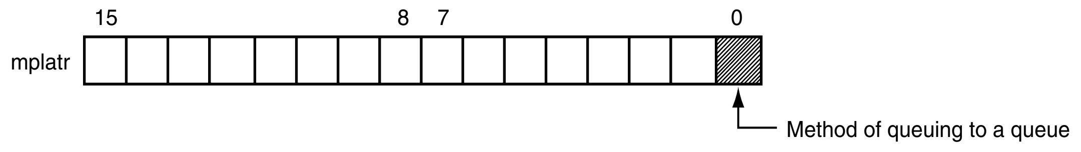
- When an ID number is not specified
  A memory pool is created based on the information specified in *pk_cmpl*.
  An ID number is allocated by RX4000 and the allocated ID number is stored into the area specified in *p_mplid*.

The following describes memory pool creation information in detail.

exinf	...	Extended information exinf is an area used for storing user-specific information for a specified memory pool. The user can use this area as necessary. Information set in exinf can be dynamically acquired by issuing the ref_mpl system call from a processing program (task/non-task).
mplatr	...	Memory pool attribute

Bit 0 .. Method of queuing to a queue

TA_TPRI(0):	Priority order
TA_TFIFO(1):	FIFO order



mplsz ... Memory pool size (bytes)

Return value

*E_OK	0	Normal termination
*E_NOMEM	–10	A memory pool management block or memory pool area cannot be allocated.
E_RSATR	–24	Invalid specification of attribute mplatr
E_PAR	–33	Invalid parameter specification – The start address of a packet storing memory pool creation information is invalid (*p_mplid* = 0). – Invalid size specification (mplsz ≤ 0).
E_ID	–35	Invalid ID number specification (maximum number of semaphores created < *mplid*)
*E_OBJ	–63	A memory pool having the specified ID number has already been created.
E_OACV	–66	An unauthorized ID number (*mplid* < 0) was specified.
E_CTX	–69	The cre_mpl system call was issued from a non-task.

del_mpl

Delete Variable-size Memory Pool (–138)

Task

Overview

Deletes a memory pool.

C format

```
#include       <stdrx.h>
ER             ercd = del_mpl(ID mplid);
```

Parameters

I/O	Parameter	Description
I	ID *mplid*;	Memory pool ID number

Explanation

This system call deletes the memory pool specified in *mplid*.
The specified memory pool is released from the control of RX4000.
The task released from the wait state (memory block wait state) by this system call has E_DLT returned as the return value of the system call (get_blk or tget_blk) that instigated transition to the wait state.

Return value

*E_OK	0	Normal termination
E_ID	–35	Invalid ID number specification (maximum number of memory pools that can be created < *mplid*)
*E_NOEXS	–52	The specified memory pool does not exist.
E_OACV	–66	An unauthorized ID number (*mplid* ≤ 0) was specified.
E_CTX	–69	The del_mpl system call was issued from a non-task.

get_blk

Get Variable-size Memory Block (–141)

Task

Overview

Acquires a memory block.

C format

```
#include      <stdrx.h>
ER            ercd = get_blk(VP *p_blk, ID mplid, INT blksz);
```

Parameters

I/O	Parameter	Description
O	VP *p_blk;	Address of area used to store the start address of the memory block
I	ID *mplid*;	Memory pool ID number
I	INT *blksz*;	Memory block size (bytes)

Explanation

This system call acquires a memory block, of the size specified in *blksz*, from the memory pool specified in *mplid* and stores its start address into the area specified in *p_blk*.
If no memory block can be acquired from a specified memory pool (when there is no free area of the requested size) upon the issue of this system call, this system call places the task in the queue of a specified memory pool before changing its state from the run state to the wait state (memory block wait state).
The memory block wait state is released when a memory block that satisfies the requested size is released by a rel_blk system call or upon the issue of a del_mpl or rel_wai system call, and the task returns to the ready state.

Caution RX4000 does not clear the memory upon acquiring a memory block. Accordingly, the contents of an acquired memory block are undefined.

Remark When a task queues in the wait queue of the specified memory pool, it is executed in the sequence (FIFO sequence or priority order sequence) specified when that memory pool was generated (during configuration or when a cre_mpl system call was issued).

Return value

*E_OK	0	Normal termination
E_PAR	–33	Invalid parameter specification – The address of the area for storing the start address of the memory block is invalid (*p_blk* = 0). – Invalid specification of memory block size (*p_blk* ≤ 0)
E_ID	–35	Invalid ID number specification (maximum number of memory pools that can be created < *mplid*)

*E_NOEXS	–52	The specified memory pool does not exist.
E_OACV	–66	An unauthorized ID number (*mplid* ≤ 0) was specified.
E_CTX	–69	Context error – The get_blk system call was issued from a non-task. – The get_blk system call was issued in the dispatch disabled state.
*E_DLT	–81	A specified memory pool was deleted using a del_mpl system call.
*E_RLWAI	–86	The memory block wait state was forcibly released by the rel_wai system call.
EV_SIGNAL	–225	The memory block wait state was forcibly released by the vsnd_sig system call.

pget_blk

Poll and Get variable-size Memory Block (–104)

Task/nontask

Overview

Acquires a memory block (polling).

C format

```
#include      <stdrx.h>
ER            ercd = pget_blk(VP *p_blk, ID mplid, INT blksz);
```

Parameters

I/O	Parameter	Description
O	VP *p_blk;	Address of area used to store the start address of a memory block
I	ID *mplid*;	Memory pool ID number
I	INT *blksz*;	Memory block size (bytes)

Explanation

This system call acquires a memory block of the size specified in *blksz* from the memory pool specified in *mplid* and stores its start address into the area specified in *p_blk*.
When this system call is issued, if no memory block can be acquired from the specified memory pool (when there is no free area of the requested size), this system call returns E_TMOUT as the return value.

Caution RX4000 does not clear the contents of memory when acquiring a memory block. Accordingly, the contents of an acquired memory block will be undefined.

Return value

*E_OK	0	Normal termination
E_PAR	–33	Invalid parameter specification – The address of the area used to store the start address of a memory block is invalid (*p_blk* = 0). – Invalid specification of memory block size (*blksz* ≤ 0)
E_ID	–35	Invalid ID number specification (maximum number of memory pools that can be created < *mplid*)
*E_NOEXS	–52	The specified memory pool does not exist.
E_OACV	–66	An unauthorized ID number (*mplid* ≤ 0) was specified.
*E_TMOUT	–85	There is no free space in the specified memory pool.

tget_blk

Get Variable-size Memory Block with Timeout (–168)

Task

Overview

Acquires a memory block (with timeout).

C format

```
#include      <stdrx.h>
ER            ercd =  tget_blk(VP *p_blk, ID mplid, INT blksz, TMO tmout);
```

Parameters

I/O	Parameter		Description
O	VP	**p_blk;*	Address of area used to store the start address of a memory block
I	ID	*mplid;*	Memory pool ID number
I	INT	*blksz;*	Memory block size (bytes)
I	TMO	*tmout;*	Wait time (basic clock cycles) TMO_POL(0): Quick return TMO_FEVR(–1): Permanent wait Value: Wait time

Explanation

This system call acquires a memory block of the size specified in *blksz* from the memory pool specified in *mplid* and stores its start address into the area specified in *p_blk*.
If a memory block cannot be acquired from a specified memory pool (when there is no free area of the requested size) when this system call is issued, this system call places the task in the queue of a specified memory pool before changing it from the run state to the wait state (memory block wait state).
The memory block wait state is released when the wait time specified in *tmout* elapses, when a memory block that satisfies the requested size is released by the rel_blk system call, or when the del_mpl or rel_wai system call is issued. Then, the task returns to the ready state.

Caution RX4000 does not clear the contents of memory upon acquiring a memory block. Accordingly, the contents of an acquired memory block will be undefined.

Remark When a task queues in the wait queue of the specified memory pool, it is executed in the sequence (FIFO sequence or priority order sequence) specified when that memory pool was generated (during configuration or when a cre_mpl system call was issued).

Return value

*E_OK	0	Normal termination
E_PAR	–33	Invalid parameter specification – The address of the area used to store the start address of memory block is invalid (*p_blk* = 0). – Invalid specification of memory block size (*blksz* ≤ 0) – Invalid wait time specification (*tmout* < TMO_FEVR)
E_ID	–35	Invalid ID number specification (maximum number of memory pools that can be created < *mplid*)
*E_NOEXS	–52	The specified memory pool does not exist.
E_OACV	–66	An unauthorized ID number (*mplid* ≤ 0) was specified.
E_CTX	–69	Context error – The tget_blk system call was issued from a non-task. – The tget_blk system call was issued in the dispatch disabled state.
*E_DLT	–81	A specified memory pool was deleted by the del_mpl system call.
*E_TMOUT	–85	Timeout elapsed.
*E_RLWAI	–86	The memory block wait state was forcibly released by a rel_wai system call.
EV_SIGNAL	–225	The memory block wait state was forcibly released by a vsnd_sig system call.

rel_blk

Release Variable-size Memory Block (–143)

Task/nontask

Overview

Returns a memory block.

C format

```
#include      <stdrx.h>
ER            ercd = rel_blk(ID mplid, VP blk);
```

Parameters

I/O	Parameter	Description
I	ID *mplid;*	Memory pool ID number
I	VP *blk;*	Start address of memory block

Explanation

This system call returns the memory block specified in *blk* to the memory pool specified in *mplid.*

If the size of the returned memory block satisfies the size requested by the task (first task in the wait queue) queuing in the specified memory pool's wait queue when this system call is issued, the memory block is transferred to the affected task (first task in the wait queue).

Consequently, the relevant task is removed from the queue, and changes from the wait state (memory block wait state) to the ready state, or from the wait_suspend state, to the suspend state.

Cautions 1. In RX4000, when a memory block is acquired, memory clear is not performed. Therefore, the contents of the acquired memory block are not definite.

2. The memory block to be returned must be the same as that specified upon the issue a get_blk, pget_blk, or tget_blk system call.

Return value

*E_OK	0	Normal termination
E_PAR	–33	Invalid parameter specification – Invalid specification of the start address of a memory block (*blk* = 0). – The memory pool specified when acquired differs from that specified upon the issue of the rel_blk system call.
E_ID	–35	Invalid ID number specification (maximum number of memory pools that can be created < *mplid*)
*E_NOEXS	–52	The specified memory pool does not exist.
*E_OBJ	–63	The returning memory block is used as a message.
E_OACV	–66	An unauthorized ID number (*mplid* ≤ 0) was specified.

ref_mpl

Refer Variable-size Memory Pool Status (–140)

Task/nontask

Overview

Acquires a memory pool information.

C format

```
#include      <stdrx.h>
ER            ercd = ref_mpl(T_RMPL *pk_rmpl, ID mplid);
```

Parameters

I/O	Parameter	Description
O	T_RMPL *pk_rmpl;*	Start address of packet used to store memory pool information
I	ID *mplid;*	Memory pool ID number

- Structure of memory pool information T_RMPL structure

```
typedef    struct     t_rmpl {
           VP         exinf;     /*  Extended information                           */
           BOOL_ID    wtsk;      /*  Existence of waiting task                      */
           INT        frsz;      /*  Total size of free area                        */
           INT        maxsz;     /*  Maximum memory block size that can be acquired */
} T_RMPL;
```

Explanation

This system call stores the memory pool information (extended information, existence of waiting tasks, etc.) for the memory pool specified in *mplid* into the packet specified in *pk_rmpl*.
Memory pool information is described in detail below.

exinf	...	Extended information
wtsk	...	Existence of waiting task FALSE(0): No waiting task Value: ID number of first task in the queue
frsz	...	Total size of free area (bytes)
maxsz	...	Maximum memory block size that can be acquired (bytes)

Return value

*E_OK	0	Normal termination
E_PAR	–33	The start address of the packet used to store memory pool information is invalid (*pk_rmpl* = 0).
E_ID	–35	Invalid ID number specification (maximum number of memory pools that can be created < *mplid*)
*E_NOEXS	–52	The specified memory pool does not exist.
E_OACV	–66	An unauthorized ID number (*mplid* ≤ 0) was specified.

12.8.7 Time management system calls

This section explains those system calls (time management system calls) that perform processing that is dependent on time.

Table 12-11 lists the time management system calls.

Table 12-11. Time Management System Calls

System call	Function
set_tim	Sets the system clock.
get_tim	Acquires the time from the system clock.
dly_tsk	Changes the task to the timeout wait state.
def_cyc	Registers a cyclically activated handler or cancels its registration.
act_cyc	Controls the activity state of a cyclically activated handler.
ref_cyc	Acquires cyclically activated handler information.

set_tim

Set Time (–83)

Task/nontask

Overview

Sets the system clock.

C format

```
#include      <stdrx.h>
ER            ercd = set_tim(SYSTIME *pk_tim);
```

Parameters

I/O	Parameter	Description
I	SYSTIME *pk_tim;	Start address of packet storing time

- Structure of system clock SYSTIME

```
typedef    struct    systime {
           UW        ltime;    /*  Time (low-order 32 bits)     */
           H         utime;    /*  Time (high-order 16 bits)    */
} SYSTIME;
```

Explanation

This system call sets the system clock to the time specified in *pk_tim*.

Return value

*E_OK	0	Normal termination
E_PAR	–33	The start address of the packet storing time is invalid (*pk_tim* = 0).

Get Time (–84)

get_tim

Task/nontask

Overview

Acquires the time from the system clock.

C format

```
#include        <stdrx.h>
ER              ercd = get_tim(SYSTIME *pk_tim);
```

Parameters

I/O	Parameter	Description
O	SYSTIME *pk_tim;	Start address of packet storing time

- Structure of system clock SYSTIME

```
typedef    struct    systime {
           UW        ltime;     /*  Time (low-order 32 bits)    */
           H         utime;     /*  Time (high-order 16 bits)   */
} SYSTIME;
```

Explanation

This system call sets the current system clock time in the packet specified in *pk_tim*.

Return value

*E_OK	0	Normal termination
E_PAR	–33	The start address of the packet used to store the time is invalid (*pk_tim* = 0).

dly_tsk

Delay Task (–85)

Task

Overview

Changes the task to the timeout wait state.

C format

```
#include      <stdrx.h>
ER            ercd = dly_tsk(DLYTIME dlytime);
```

Parameters

I/O	Parameter	Description
I	DLYTIME dlytim;	Delay time (basic clock cycles)

Explanation

This system call changes the state of the task from the run state to the wait state (timeout wait state) by the delay time specified in *dlytim*.

The timeout wait state is released upon the elapse of the delay specified in *dlytim* or when the rel_wai system call is issued. Then, the task returns to the ready state.

Caution The timeout wait state is released by neither the wup_tsk or ret_wup system call.

Return value

*E_OK	0	Normal termination
E_PAR	–33	Invalid specification of delay (*dlytim* < 0)
E_CTX	–69	Context error – The dly_tsk system call was issued from a non-task. – The dly_tsk system call was issued in the dispatch disabled state.
*E_RLWAI	–86	The timeout wait state was forcibly released by the issue of a rel_wai system call.

def_cyc

Define Cyclic Handler (–90)

Task/nontask

Overview

Registers a cyclically activated handler or cancels its registration.

C format

```
#include       <stdrx.h>
ER             ercd = def_cyc(HNO cycno, T_DCYC *pk_dcyc);
```

Parameters

I/O	Parameter	Description
I	HNO *cycno*;	Specification number of cyclically activated handler
I	T_DCYC **pk_dcyc*;	Start address of packet storing the cyclically activated handler registration information

- Structure of cyclically activated handler registration information T_DCYC

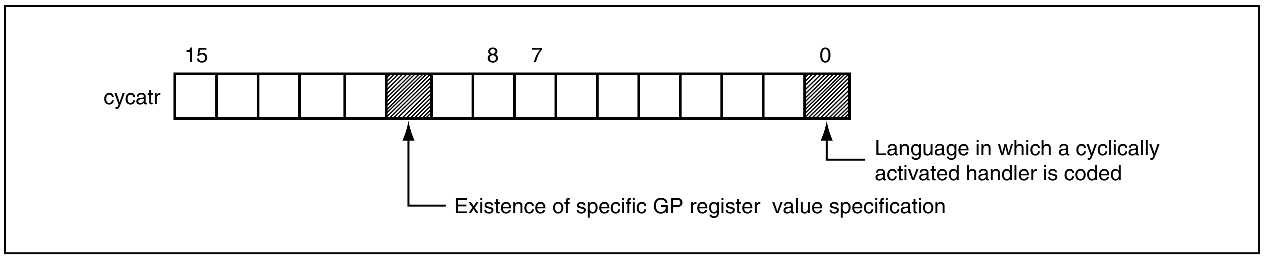
```
typedef   struct    t_dcyc {
          VP        exinf;   /*  Extended information                                     */
          ATR       cycatr;  /*  Attribute of cyclically activated handler                */
          FP        cychdr;  /*  Activation address of cyclically activated handler       */
          UINT      cycact;  /*  Initial activity state of cyclically activated handler   */
          CYCTIME   cyctim;  /*  Activation time interval of cyclically activated handler */
          VP        gp;      /*  Specific GP register value of cyclically activated handler */
} T_DCYC;
```

Explanation

This system call uses the information specified in *pk_dcyc* to register the cyclically activated handler having the specification number specified in *cycno*.
The cyclically activated handler registration information is described in detail below.

exinf ... Extended information
exinf is an area used for storing user-specific information on a specified task. The user can use this area as necessary.
Information set in exinf can be dynamically acquired by issuing an ref_cyc system call from a processing program (task/non-task).

cycatr ... Attribute of cyclically activated handler
- Bit 0 .. Language in which the cyclically activated handler is encoded
  - TA_ASM(0): Assembly language
  - TA_HLNG(1): C
- Bit 10 .. Existence of specific GP register value specification
  - TA_DPID(1): Specifies a specific GP register value.


15
8
7
0
cycatr
Language in which a cyclically activated handler is coded
Existence of specific GP register value specification

cychdr	...	Activation address of cyclically activated handler
cycact	...	Initial activity state of cyclically activated handler
		TCY_OFF(0): The initial activity state is OFF
		TCY_ON(1): The initial activity state is ON
cyctim	...	Activation time interval of cyclically activated handler (basic clock cycles)
gp	...	Specific GP register value for cyclically activated handler

When this system call is issued, if a cyclically activated handler corresponding to a specified specification number is already registered, this system call does not handle this as an error and newly registers the specified cyclically activated handler.

If this system call is issued with NADR(–1) set in the area specified in *pk_dcyc*, the registration of the cyclically activated handler specified in *cycno* is canceled.

Remark If the value 1 of bit 10 of cycatr is other than TA_DPID, the contents of gp will be meaningless.

Return value

*E_OK	0	Normal termination
E_RSATR	–24	Invalid specification of attribute cycatr
E_PAR	–33	Invalid parameter specification

- Invalid specification of specification number (*cycno* ≤ 0, maximum number of cyclically activated handlers that can be registered < *cycno*)
- The start address of the packet storing cyclically activated handler registration information is invalid (*pk_dcyc* = 0).
- Invalid specification of activation address (cychdr = 0)
- Invalid specification of initial activity state cycact
- Invalid specification of activation time interval (cyctim ≤ 0)

act_cyc

Activate Cyclic Handler (–94)

Task/nontask

Overview

Controls the activity state of a cyclically activated handler.

C format

```
#include      <stdrx.h>
ER            ercd = act_cyc(HNO cycno, UINT cycact);
```

Parameters

I/O	Parameter	Description
I	HNO *cycno;*	Specification number of cyclically activated handler
I	UINT *cycact;*	Specification of activity state and cycle counter TCY_OFF(0): Changes the activity state to the OFF state. TCY_ON(1): Changes the activity state to the ON state. TCY_INI(2): Initializes the cycle counter.

Explanation

This system call changes the activity state of the cyclically activated handler specified in *cycno* to the state specified in *cycact*. The specification format of *cycact* is described below.

- *cycact* = TCY_OFF
  Changes the activity state of the cyclically activated handler to the OFF state. Even when the activation time is reached, the cyclically activated handler is not activated.

  Caution Even when the activity state of the cyclically activated handler is off, RX4000 increments the cycle counter.

- *cycact*=TCY_ON
  Changes the activity state of a cyclically activated handler to the ON state. When the activation time is reached, the specified cyclically activated handler is activated.

- *cycact*=TCY_INI
  Initializes the cycle counter of the specified cyclically activated handler.

- *cycact* = (TCY_ON|TCY_INI)
  Changes the activity state of the specified cyclically activated handler to the ON state before initializing the cycle counter.
  When the activation time is reached, the specified cyclically activated handler is activated.

Return value

*E_OK	0	Normal termination
E_PAR	–33	Invalid parameter specification – The specification number of the cyclically activated handler is invalid (*cycno* ≤ 0, maximum number of cyclically activated handlers that can be registered < *cycno*)). – Invalid specification of activity state or cycle counter *cycact*
*E_NOEXS	–52	The specified cyclically activated handler is not registered.

ref_cyc

Refer Cyclic Handler Status (–92)

Task/nontask

Overview

Acquires cyclically activated handler information.

C format

```
#include      <stdrx.h>
ER            ercd = ref_cyc(T_RCYC *pk_rcyc, HNO cycno);
```

Parameters

I/O	Parameter	Description
O	T_RCYC *pk_rcyc;	Start address of packet used to store cyclically activated handler information
I	HNO cycno;	Specification number of cyclically activated handler

- Structure of cyclically activated handler information T_RCYC

```
typedef    struct     t_rcyc {
           VP         exinf;     /*  Extended information      */
           CYCTIME    lfttim;    /*  Remaining time            */
           UINT       cycact;    /*  Current activity state    */
} T_RCYC;
```

Explanation

This system call stores the cyclically activated handler information (extended information, remaining time, etc.) of the cyclically activated handler specified in *cycno* into the packet specified in *pk_rcyc*.
Cyclically activated handler information is described in detail below.

exinf	...	Extended information
lfttim	...	Time remaining until the cyclically activated handler is next activated (basic clock cycles)
cycact	...	Current activity state

TCY_OFF(0):	Activity state is OFF.
TCY_ON(1):	Activity state is ON.

Return value

*E_OK	0	Normal termination
E_PAR	–33	Invalid parameter specification – The start address of the packet used to store cyclically activated handler information is invalid (*pk_rcyc* = 0). – The specification number of the cyclically activated handler is invalid (*cycno* ≤ 0, maximum number of cyclically activated handlers that can be registered < *cycno*)).
*E_NOEXS	–52	The specified cyclically activated handler is not registered.

12.8.8 System management system calls

This section explains those system calls (system management system calls) that perform processing that is dependent on the system.

Table 12-12 lists the system management system calls.

Table 12-12. System Management System Calls

System call	Function
get_ver	Acquires RX4000 version information.
ref_sys	Acquires system information.
def_svc	Registers an extended SVC handler or cancels its registration.
viss_svc	Calls an extended SVC handler.
def_exc	Registers and cancels registration of exception handlers.

get_ver

Get Version Information (–16)

Task/nontask

Overview

Acquires RX4000 version information.

C format

```
#include      <stdrx.h>
ER            ercd = get_ver(T_VER *pk_ver);
```

Parameters

I/O	Parameter	Description
O	T_VER *pk_ver;*	Start address of packet used to store version information

- Structure of version information T_VER

```
typedef    struct    t_ver {
           UH        maker;      /*  OS maker                                          */
           UH        id;         /*  OS format                                         */
           UH        spver;      /*  Specification version                             */
           UH        prver;      /*  OS version                                        */
           UH        prno[4];    /*  Product number, production management information */
           UH        cpu;        /*  CPU information                                   */
           UH        var;        /*  Variation descriptor                              */
} T_VER;
```

Explanation

This system call stores the RX4000 version information (OS maker, OS format, etc.) into the packet specified in *pk_ver*.
Version information is described in detail below.

maker	...	OS maker	
		H'000d:	NEC
id	...	OS format	
		H'0000:	Not used
spver	...	Specification version	
		H'5302:	μITRON3.0 Ver. 3.02
prver	...	OS product version	
		H'0300:	RX4000 Ver. 3.00

prno[4]	...	Product number/product management information	
		Undefined:	Serial number of delivery product (each unit has a unique number)
cpu	...	CPU information	
		H'0d21:	VR4100
var	...	Variation descriptor	
		H'c000:	μITRON level E, for single processor use, virtual storage not supported, MMU not supported, file not supported

Return value

*E_OK	0	Normal termination
E_PAR	–33	Start address of the packet used to store version information is invalid (*pk_ver* = 0).

ref_sys

Refer System Status (–12)

Task/nontask

Overview

Acquires system information.

C format

```
#include      <stdrx.h>
ER            ercd = ref_sys(T_RSYS *pk_rsys);
```

Parameter

I/O	Parameter	Description
O	T_RSYS *pk_rsys;	Start address of packet used to store system information

- Structure of system information T_RSYS

```
typedef   struct   t_rsys {
          INT      sysstat;    /*  System state     */
} T_RSYS;
```

Explanation

This system call stores the current value of dynamically-changing system information (system state) into the packet specified in *pk_rsys*.
System information is described in detail below.

sysstat ... System state

TTS_TSK(0):	Task processing is being performed. Dispatch processing is enabled.
TTS_DDSP(1):	Task processing is being performed. Dispatch processing is disabled.
TTS_LOC(3):	Task processing is being performed. The acceptance of maskable interrupts and dispatch processing is disabled.
TTS_INDP(4):	Processing of a non-task (interrupt handler, cyclically activated handler, etc.) is being performed.

Return value

*E_OK	0	Normal termination
E_PAR	–33	The start address of a packet used to store system information is invalid (*pk_rsys* = 0).

def_svc

Define Supervisor Call Handler (–9)

Task/nontask

Overview

Registers an extended SVC handler or cancels its registration.

C format

```
#include      <stdrx.h>
ER            ercd = def_svc(FN s_fncd, T_DSVC *pk_dsvc);
```

Parameters

I/O	Parameter	Description
I	FN *s_fncd;*	Extended function code of extended SVC handler
I	T_DSVC **pk_dsvc;*	Start address of packet storing the extended SVC handler registration information

- Structure of extended SVC handler registration information T_DSVC

```
typedef   struct   t_dsvc {
          ATR      svcatr;    /*  Attribute of extended SVC handler                     */
          FP       svchdr;    /*  Activation address of extended SVC handler            */
          VP       gp;        /*  Specific GP register value for extended SVC handler   */
} T_DSVC;
```

Explanation

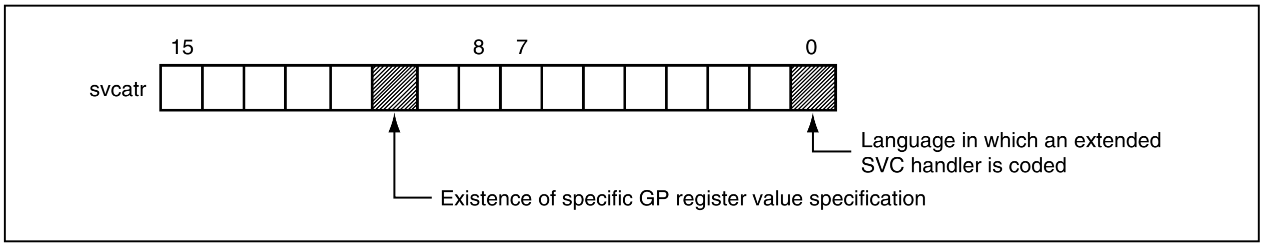
This system call uses information specified in *pk_dsvc* to register the extended SVC handler having the extended function code specified in *s_fncd*.
Extended SVC handler registration information is described in detail below.

svcatr ... Attribute of extended SVC handler
Bit 0 ... Language in which the extended SVC handler is coded

TA_ASM(0):	Assembly language
TA_HLNG(1):	C

Bit 10 ... Existence of specific GP register value specification

TA_DPID(1):	Specifies a specific GP register value.


15
8
7
0
svcatr
Language in which an extended SVC handler is coded
Existence of specific GP register value specification

svchdr	...	Activation address of extended SVC handler
gp	...	Specific GP register value of extended SVC handler

When this system call is issued, if an extended SVC handler corresponding to a specified interrupt level has already been registered, this system call does not handle this as an error and newly registers the specified extended SVC handler.

When this system call is issued, if NADR(–1) is set in the area specified in *pk_dsvc*, the registration of the extended SVC handler specified in *s_fncd* is canceled.

Remark When the value 1 of bit 10 of svcatr is other than TA_DPID, the contents of gp are meaningless.

Return value

*E_OK	0	Normal termination
E_RSATR	–24	Invalid specification of attribute svcatr
E_PAR	–33	Invalid parameter specification – Invalid specification of extended function code (*s_fncd* ≤ 0, maximum number of extended SVC handlers that can be registered < *s_fncd*) – The start address of the packet storing the extended SVC handler registration information is invalid (*pk_dsvc* = 0). – Invalid specification of activation address (svchdr = 0)

viss_svc

Issued Supervisor Call Handler (–250)

Task/nontask

Overview

Calls an extended SVC handler.

C format

```
#include      <stdrx.h>
ER            ercd = viss_svc(FN s_fncd, VW prm1, VW prm2, VW prm3);
```

Parameters

I/O	Parameter		Description
I	FN	*s_fncd;*	Extended function code of extended SVC handler
I	VW	*prm1;*	Parameter 1 passed to extended SVC handler
I	VW	*prm2;*	Parameter 2 passed to extended SVC handler
I	VW	*prm3;*	Parameter 3 passed to extended SVC handler

Explanation

This system call calls the extended SVC handler having the extended function code specified in *s_fncd*.

Remark When this system call is used to call an extended SVC handler, the interface library for the extended SVC handlers need not be coded.

Return value

*E_OK	0	Normal termination
E_PAR	–33	Invalid specification of extended function code (*s_fncd* $\leq$ 0, maximum number of extended SVC handlers that can be registered < *s_fncd*)
Others		Return value from extended SVC handler

def_exc

Define Exception Handler (–11)

Task/nontask

Overview

Registers and cancels registration of exception handlers.

C format

```
#include      <stdrx.h>
ER            ercd = def_exc(INT exckind, T_DEXC *pk_dexc);
```

Parameters

I/O	Parameter	Description
I	INT *exckind;*	Type of Exception TEK_CPU (0) : CPU exception TEK_SYS (1) : System call exception
I	T_DEXC **pk_dexc;*	Start address of the packet where exception handler registration information is stored.

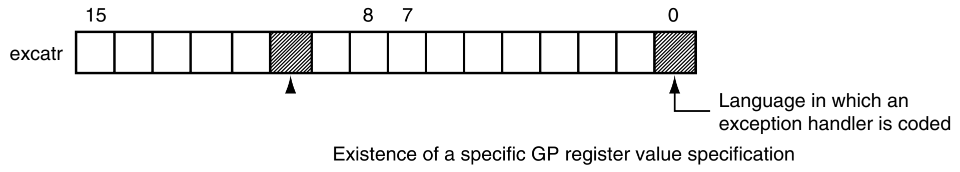
- Structure of exception handler registration information T_DEXC

```
typedef   struct   t_desc {
          ATR      excatr;    /*  Exception handler attribute                       */
          FP       exchdr;    /*  Exception handler starting address                */
          VP       gp;        /*  Specific GP register value for exception handler  */
} T_DEXC;
```

Explanation

Based on the information specified in *pk_dexc*, an exception handler which corresponds to the type of exception specified in *exckind* is registered. Details of exception handler registration information are shown below.

excatr ... Exception handler attribute
- Bit 0 ... Exception handler description language
  - TA_ASM(0): Assembly language
  - TA_HLNG(1): C
- Bit 10 ... Existence of specific GP register value specification
  - TA_DPID(1): Specifies a specific GP register value


15
8
7
0
excatr
Language in which an exception handler is coded
Existence of a specific GP register value specification

exchdr	...	Exception handler’s start address
gp	...	Specific GP register value for exception handler

If an exception handler corresponding to the type of exception which is the object of this system call is already registered when this system call is issued, the exception handler specified by this system call is newly registered.

Also, if NADR (-1) is specified in the area specified by *pk_dsvc* when this system call is issued, registration of the exception handler corresponding to the type of exception specified in *exckind* is canceled.

Remark When the value 1 of bit 10 of excatr is other than TA_DPID, the contents of gp are meaningless.

Return value

*E_OK	0	Ended normally
E_RSATR	–24	Invalid specification of attribute excatr
E_PAR	–33	The parameter specification is invalid – The specification for the type of exception is invalid (*exckind* <0, 1 < *exckind*). – The start address of the packet where the exception handler registration information is stored is invalid (*pk_dexc* = 0). – The starting address specification is invalid (exchdr = 0).

APPENDIX A PROGRAMMING METHODS

This appendix explains how to write processing programs when CodeWarrior (Metroworks Corporation) or C Cross MIPE Compiler, produced by Green Hills Software Inc., is being used.

A.1 Overview

In RX4000, processing programs are classified according to purpose, as shown below.

- Task
  The minimum unit of a processing program which can be executed by RX4000.

- Interrupt handler
  A routine dedicated to interrupt handling. When an interrupt occurs, this handler is activated upon the completion of the preprocessing by RX4000 (such as saving the contents of the registers or switching the stack).

- Cyclically activated handler
  A routine dedicated to cyclic processing. Every time the specified time elapses, this handler is activated immediately. This routine is handled independently of tasks. At the activation time, therefore, the processing of a task currently being executed is canceled even if that task has the highest priority relative to all other tasks in the system. Then, control is passed to the cyclically activated handler.
  A cyclically activated handler incurs a smaller overhead before the start of execution, relative to any other cyclic processing programs written by the user.

- Extended SVC handler
  A function registered by the user as an extended system call.

- Task-associated handler
  This is an exclusive routine for performing processing with respect to external occurrence (signals) generated by each respective task. Since it is positioned as an extension of the task which generated the occurrence, it is executed in the context of the specified task. Also, this handler has the same priority level as the specified task and is scheduled at the same level as the task.

These processing programs have their own basic formats according to the general conventions or conventions to be applied when RX4000 is used.

A.2 Keywords

The character strings listed below are reserved as keywords for the configurator. These strings shall not, therefore, be used for other purposes.

IO	MIO	RAM	ROM	VOID
action	addr	auto	cache	clktim
cychdr	default	defstk	eventflag	exchdr
exinf	flgsvc	gp	id	idlehdr
initcnt	inithdr	initptn	inthdr	intr
intstk	intsvc	intvl	kernel	kind
lang	level	mailbox	map	mask
maxcnt	maxcyc	maxflg	maxmbx	maxmpl
maxpri	maxsem	maxsvc	maxtsk	mbxsvc
memory	memorypool	memtype	mplsvc	name
nouse	number	pagesize	pool	pri
prot	segment	semaphore	semsvc	sighdr
sigsvc	size	sncsvc	staaddr	stack
task	timsvc	tsksvc	type	user
waiopt	All system call names			

A.3 Reserved Words

The character strings listed below are reserved as external symbols for RX4000. These strings shall not, therefore, be used for other purposes.

sit	_ _rx_start

A.4 Tasks

A.4.1 When CodeWarrior is used

When describing a task in C, describe it as an void-type function having one INT-type argument.

As an argument (*stacd*), the activation code specified when the sta_tsk system call is issued is set.

Figure A-1 shows the task description format (in C) when CodeWarrior is used.

Figure A-1. Task Description Format (C)

```
#include   <stdrx.h>

void
func_task(INT stacd)
{
          /* Processing of task func_task */
           ............................
           ............................
           ............................

          /* Termination of task func_task */
          ext_tsk();
}
```

When describing a task in assembly language, describe it as a function conforming to the function call conventions of CodeWarrior.

As an argument (r4 register), the activation code specified when the sta_tsk system call is issued is set.

Figure A-2 shows the task description format (in assembly language) when CodeWarrior is used.

Figure A-2. Task Description Format (Assembly Language)

```
.include    "stdrx.inc"

            .text
            .align        4
            .globl        _func_task
_func_task:
            # Processing of task func_task
             .........................
             .........................
             .........................

            # Termination of task func_task
            jr            _ext_tsk
```

A.4.2 When C Cross MIPE Compiler is used

When describing a task in C, describe it as an void-type function having one INT-type argument.

As an argument (stacd), the activation code specified upon the issue of the sta_tsk system call is set.

Figure A-3 shows the task description format (in C) when C Cross MIPE Compiler is used.

Figure A-3. Task Description Format (C)

```
#include   <stdrx.h>

void
func_task(INT stacd)
{
            /* Processing of task func_task */
             ............................
             ............................
             ............................

            /* Termination of task func_task */
            ext_tsk();
}
```

When describing a task in assembly language, describe it as a function conforming to the function call conventions of C Cross MIPE Compiler.

As an argument (r4 register), the activation code specified upon the issue of the sta_tsk system call is set.

Figure A-4 shows the task description format (in assembly language) when C Cross MIPE Compiler is used.

Figure A-4. Task Description Format (Assembly Language)

```
#include  <stdrx.h>

          .text
          .align     4
          .globl     _func_task
_func_task:
          #  Processing of task  func_task
          ........................
          ........................
          ........................

          #  Termination of task  func_task
          jr         _ext_tsk
```

Caution When describing a task in assembly language, specify ".s" as the file extension.

A.5 Interrupt Handler

A.5.1 When CodeWarrior is used

When describing an interrupt handler in C, describe it as an INT-type function having no argument.
Figure A-5 shows the description format of an interrupt handler (in C) when CodeWarrior is used.

Figure A-5. Description Format of Interrupt Handler (C)

```
#include   <stdrx.h>

INT
func_inthdr()
{
          /*  Processing of interrupt handler func_inthdr */
           ............................................................
           ............................................................
           ............................................................

          /* Return processing from interrupt handler func_inthdr */
          return(TSK_NULL);
}
```

Remark An interrupt handler is a subroutine called by interrupt processing in the nucleus. Therefore, when an interrupt handler is described, an instruction for branching to the interrupt handler need not be set for the handler address to which the processor passes control upon the occurrence of an interrupt.

When describing an interrupt handler in assembly language, describe it as a function conforming to the function call conventions of CodeWarrior.

Figure A-6 shows the description format of an interrupt handler (in assembly language) when CodeWarrior is used.

Figure A-6. Description Format of Interrupt Handler (Assembly Language)

```
.include   "stdrx.inc"

           .text
           .align       4
           .globl       _func_inthdr
_func_inthdr:
           # Processing of interrupt handler func_inthdr
           ..........................................................
           ..........................................................
           ..........................................................

           # Return processing from interrupt handler func_inthdr
           li           TSK_NULL, r2
           jr           $ra
```

Remark An interrupt handler is a subroutine called by interrupt processing in the nucleus. Therefore, when an interrupt handler is described, an instruction for branching to the interrupt handler need not be set for the handler address to which the processor passes control upon the occurrence of an interrupt.

A.5.2 When C Cross MIPE Compiler is used

When describing an interrupt handler in C, describe it as an INT-type function having no argument.

Figure A-7 shows the description format of an interrupt handler (in C) when C Cross MIPE Compiler is used.

Figure A-7. Description Format of Interrupt Handler (C)

```
#include <stdrx.h>

INT
func_inthdr()
{
        /* Processing of interrupt handler func_inthdr */
         ..........................................................
         ..........................................................
         ..........................................................

        /* Return processing from interrupt handler func_inthdr */
        return(TSK_NULL);
}
```

Remark An interrupt handler is a subroutine called by interrupt processing in the nucleus. Therefore, when an interrupt handler is described, an instruction for branching to the interrupt handler need not be set for the handler address to which the processor passes control upon the occurrence of an interrupt.

When describing an interrupt handler in assembly language, describe it as a function conforming to the function call conventions of C Cross MIPE Compiler.

Figure A-8 shows the description format of an interrupt handler (in assembly language) when C Cross MIPE Compiler is used.

Figure A-8. Description Format of Interrupt Handler (Assembly Language)

```
#include   <stdrx.h>

           .text
           .align     4
           .globl     _func_inthdr
_func_inthdr:
           # Processing of interrupt handler func_inthdr
            .....................................................
            .....................................................
            .....................................................

           # Return processing from interrupt handler func_inthdr
           li         TSK_NULL, r2
           jr         $ra
```

Remark An interrupt handler is a subroutine called by interrupt processing in the nucleus. Therefore, when an interrupt handler is described, an instruction for branching to the interrupt handler need not be set for the handler address to which the processor passes control upon the occurrence of an interrupt.

Caution When describing an interrupt handler in assembly language, specify ".s" as the file extension.

A.6 Cyclically Activated Handler

A.6.1 When CodeWarrior is used

When describing a cyclically activated handler in C, describe it as an INT-type function having no argument. Figure A-9 shows the description format of a cyclically activated handler (in C) when Code Warrior is used.

Figure A-9. Description Format of Cyclically Activated Handler (C)

```
#include   <stdrx.h>

INT
func_cychdr()
{
          /* Processing of cyclically activated handler func_cychdr */
           ......................................................
           ......................................................
           ......................................................

          /* Return processing from cyclically activated handler func_cychdr */
          ret_tmr();
}
```

Remark A cyclically activated handler is a subroutine called by system clock processing in the nucleus.

When describing a cyclically activated handler in assembly language, describe it as a function conforming to the function call conventions of CodeWarrior.

Figure A-10 shows the description format of a cyclically activated handler (in assembly language) when CodeWarrior is used.

Figure A-10. Description Format of Cyclically Activated Handler (Assembly Language)

```
.include   "stdrx.inc"

           .text
           .align       4
           .globl       _func_cychdr
_func_cychdr:
           # Processing of cyclically activated handler func_cychdr
            ................................................
            ................................................
            ................................................

           # Return processing from cyclically activated handler func_cychdr
           jr           $ra
```

Remark A cyclically activated handler is a subroutine called by system clock processing in the nucleus.

A.6.2 When C Cross MIPE Compiler is used

When describing a cyclically activated handler in C, describe it as an INT-type function having no argument.

Figure A-11 shows the description format of a cyclically activated handler (in C) when C Cross MIPE Compiler is used.

Figure A-11. Description Format of Cyclically Activated Handler (C)

```
#include   <stdrx.h>

INT
func_cychdr()
{
           /*  Processing of cyclically activated handler func_cychdr */
            ......................................................
            ......................................................
            ......................................................

           /* Return processing from cyclically activated handler func_cychdr */
           ret_tmr();
}
```

Remark A cyclically activated handler is a subroutine called by system clock processing in the nucleus.

When describing a cyclically activated handler in assembly language, describe it as a function conforming to the function call conventions of C Cross MIPE Compiler.

Figure A-12 shows the description format of a cyclically activated handler (in assembly language) when C Cross MIPE Compiler is used.

Figure A-12. Description Format of Cyclically Activated Handler (Assembly Language)

```
#include   <stdrx.h>

           .text
           .align     4
           .globl     _func_cychdr
_func_cychdr:
           # Processing of cyclically activated handler func_cychdr
            ..............................................
            ..............................................
            ..............................................

           # Return processing from cyclically activated handler func_cychdr
           jr         $ra
```

Remark A cyclically activated handler is a subroutine called by system clock processing in the nucleus.

Caution When describing a cyclically activated handler in assembly language, specify ".s" as the file extension.

A.7 Extended SVC Handler

A.7.1 When CodeWarrior is used

When describing an extended SVC handler in C, describe it as an INT-type function.

Figure A-13 shows the description format of an extended SVC handler (in C) when CodeWarrior is used.

Figure A-13. Description Format of Extended SVC Handler (C)

```
#include   <stdrx.h>

INT
func_svchdr(VW prm1, VW prm2, VW prm3)
{
           int      ret;

           /*  Processing of extended SVC handler func_svchdr */
            .................................................
            .................................................
            .................................................

           /* Return processing from extended SVC handler func_svchdr */
           return(INT ret);
}
```

When describing an extended SVC handler in assembly language, describe it as a function conforming to the function call conventions of CodeWarrior.

Figure A-14 shows the description format of an extended SVC handler (in assembly language) when CodeWarrior is used.

Figure A-14. Description Format of Extended SVC Handler (Assembly Language)

```
.include   "stdrx.inc"

           .text
           .align       4
           .globl       _func_svchdr
_func_svchdr:
           # Processing of extended SVC handler func_svchdr
            ..............................................
            ..............................................
            ..............................................

           # Return processing from extended SVC handler func_svchdr
           li           ret, r2
           jr           $ra
```

A.7.2 When C Cross MIPE Compiler is used

When describing an extended SVC handler in C, describe it as an INT-type function.

Figure A-15 shows the description format of an extended SVC handler (in C) when C Cross MIPE Compiler is used.

Figure A-15. Description Format of Extended SVC Handler (C)

```
#include   <stdrx.h>

INT
func_svchdr(VW prm1, VW prm2, VW prm3)
{
          int       ret;

          /*  Processing of extended SVC handler func_svchdr */
           ..................................................
           ..................................................
           ..................................................

          /* Return processing from extended SVC handler func_svchdr */
          return(INT ret);
}
```

When describing an extended SVC handler in assembly language, describe it as a function conforming to the function call conventions of C Cross MIPE Compiler.

Figure A-16 shows the description format of an extended SVC handler (in assembly language) when C Cross MIPE Compiler is used.

Figure A-16. Description Format of Extended SVC Handler (Assembly Language)

```
#include    <stdrx.h>

            .text
            .align     4
            .globl     _func_svchdr
_func_svchdr:
            #  Processing of extended SVC handler func_svchdr
             ...............................................
             ...............................................
             ...............................................

            #  Return processing from extended SVC handler func_svchdr
            li         ret, r2
            jr         $ra
```

Caution When describing an extended SVC handler in assembly language, specify ".s" as the file extension.

A.8 Task-Associated Handler

A.8.1 When CodeWarrior is used

When describing the task-associated handler is described in C, describe it as a void-type function.

Figure A-17 shows the description format of a task-associated handler (in C) when CodeWarrior is used.

Figure A-17. Description Format of Task-Associated Handler (C)

```
#include   <stdrx.h>

void
func_sighdr(VW prm1, VW prm2, VW prm3)
{

           /* Processing of task-associated handler func_sighdr */
           ...................................................
           ...................................................
           ...................................................

           /* Return processing from task-associated handler func_sighdr.*/
           return;
}
```

When describing a task-associated handler in assembly language, describe it as a function conforming to the function call conventions of CodeWarrior.

Figure A-18 shows the description format of a task-associated handler (in assembly language) when Codewarrior is used.

Figure A-18. Task-Associated Handler Description Format (Assembly Language)

```
#include   "stdrx.inc"

           .text
           .align       4
           .globl       _func_sighdr
_func_sighdr:
           #  Processing of task-associated handler func_sighdr
            ...............................................
            ...............................................
            ...............................................

           #  Return processing from task-associated handler func_sighdr
           jr           $ra
```

A.8.2 When C Cross MIPE Compiler is used

When describing a task-associated handler in C, describe it as a void-type function.

Figure A-19 shows the description format of a task-associated handler (in C) when C Cross MIPE Compiler is used.

Figure A-19. Task-Associated Handler Description Format (C)

```
#include   <stdrx.h>

void
func_sighdr(VW prm1, VW prm2, VW prm3)
{

          /* Processing of task-associated handler func_sighdr */
           ..................................................
           ..................................................
           ..................................................

          /* Return processing from task-associated handler func_sighdr.*/
          return;
}
```

When describing a task-associated handler in assembly language, describe it as a function conforming to the function call convention of C Cross MIPE Compiler.

Figure A-20 shows the description format of task-associated handler (in assembly language) when C Cross MIPE Compiler is used.

Figure A-20. Description Format of Task-Associated Handler (Assembly Language)

```
#include   <stdrx.h>

           .text
           .align     4
           .globl     _func_sighdr
_func_sighdr:
           # Processing of task-associated handler func_sighdr
            ..............................................
            ..............................................
            ..............................................

           # Return processing from task-associated handler func_sighdr
           jr         $ra
```

Caution When describing a task-associated handler in assembly language, specify ".s" as the file extension.

APPENDIX B INDEX

NEC

Facsimile Message

Although NEC has taken all possible steps to ensure that the documentation supplied to our customers is complete, bug free and up-to-date, we readily accept that errors may occur. Despite all the care and precautions we've taken, you may encounter problems in the documentation. Please complete this form whenever you'd like to report errors or suggest improvements to us.

From:

Name

Company

Tel. FAX

Address

Thank you for your kind support.

North America NEC Electronics Inc. Corporate Communications Dept. Fax: +1-800-729-9288 +1-408-588-6130	**Hong Kong, Philippines, Oceania** NEC Electronics Hong Kong Ltd. Fax: +852-2886-9022/9044	**Asian Nations except Philippines** NEC Electronics Singapore Pte. Ltd. Fax: +65-250-3583
Europe NEC Electronics (Europe) GmbH Technical Documentation Dept. Fax: +49-211-6503-274	**Korea** NEC Electronics Hong Kong Ltd. Seoul Branch Fax: +82-2-528-4411	**Japan** NEC Semiconductor Technical Hotline Fax: +81- 44-435-9608
South America NEC do Brasil S.A. Fax: +55-11-6462-6829	**Taiwan** NEC Electronics Taiwan Ltd. Fax: +886-2-2719-5951	

I would like to report the following error/make the following suggestion:

Document title:

Document number: Page number:

If possible, please fax the referenced page or drawing.

Document Rating	Excellent	Good	Acceptable	Poor
Clarity	❑	❑	❑	❑
Technical Accuracy	❑	❑	❑	❑
Organization	❑	❑	❑	❑

CS 01.2