User's Manual

RX850

Real-Time Operating System

Basics

Target device
V850 family™
Target real-time OS
RX850 Ver. 3.13 or later

Document No. U13430EJ2V1UM00 (2nd edition)
Date Published October 2001 J CP(K)

[MEMO]

SUMMARY OF CONTENTS

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

MAJOR REVISIONS IN THIS EDITION

Pages	Description
p.22	Modification of description in **Section 1.7**
p.23	Modification of description in **Section 1.8**
p.26	Correction of **Figure 1-1**
p.27	Correction of **Figure 1-2**
p.207	Addition of **Appendix A.8**
p.217	Addition of **Appendix C**

The mark ★ shows major revised points.

[MEMO]

PREFACE

Users	This manual is intended for those users who design and develop application systems of the V850 family.
Purpose	This manual explains the functions of the RX850.
Organization	This manual includes the following:

- Overview
- Nucleus
- Task management function
- Synchronous communication functions
- Interrupt management function
- Memory pool management function
- Time management function
- Scheduler
- System initialization
- System calls

How to read this manual It is assumed that the readers of this manual have general knowledge on electric engineering, logic circuits, microcontrollers, the C language, and assembler.

To learn the hardware functions of the V850 family
→ Refer to **User's Manual, Hardware** for each product.

To learn the instruction functions of the V850 family
→ Refer to **User's Manual, Architecture** for each product.

Notation

Data signification	:	Left: higher digit, right: lower digit		
Note	:	Explanation of item indicated in the text		
Caution	:	Information to which the user should afford special attention		
Remark	:	Supplementary information		
Numeric value	:	Binary	:	XXXX or XXXXB
		Decimal	:	XXXX
		Hexadecimal	:	0xXXXX

Units for representing powers of 2 (address space or memory space):

K (kilo)	:	2^{10} = 1,024
M (mega)	:	2^{20} = $1{,}024^2$

Related documents When using this manual, also refer to the following documents. Some related documents may be preliminary versions. Note, however, that whether a related document is preliminary is not indicated in this document.

Documents related to development tools (User's manual)

Document name		Document No.
IE-703002-MC (In-circuit emulator for V851$^{\text{TM}}$, V852$^{\text{TM}}$, V853$^{\text{TM}}$, V854$^{\text{TM}}$, V850/SA1$^{\text{TM}}$, V850/SB1$^{\text{TM}}$, V850/SB2$^{\text{TM}}$, V850/SV1$^{\text{TM}}$)		U11595E
IE-703003-MC-EM1 (Peripheral I/O board for V853)		U11596E
IE-703008-MC-EM1 (Peripheral I/O board for V854)		U12420E
IE-703017-MC-EM1 (Peripheral I/O board for V850/SA1)		U12898E
IE-703037-MC-EM1 (Peripheral I/O board for V850/SB1, V850/SB2)		U14151E
IE-703040-MC-EM1 (Peripheral I/O board for V850/SV1)		U14337E
IE-703102-MC (In-circuit emulator for V850E/MS1$^{\text{TM}}$)		U13875E
IE-703102-MC-EM1, IE-703102-MC-EM1-A (Peripheral I/O board for V850E/MS1)		U13876E
IE-V850E-MC (In-circuit emulator for V850E/IA1$^{\text{TM}}$), IE-V850E-MC-A (In-circuit emulator for V850E1 (NB85E core), V850E/MA1$^{\text{TM}}$)		U14487E
IE-V850E-MC-EM1-A (Peripheral I/O board for V850E1 (NB85E core))		To be prepared
IE-V850E-MC-EM1-B, IE-V850E-MC-MM2 (Peripheral I/O board for V850E1 (NB85E core))		U14482E
IE-703107-MC-EM1 (Peripheral I/O board for V850E1/MA1)		U14481E
IE-703116-MC-EM1 (Peripheral I/O board for V850E1/IA1)		To be prepared
V800 Series$^{\text{TM}}$ Development Tool (for 32-bit) Application Note Tutorial Guide Windows-based		U14218E
CA850 (C compiler package)	Operation	U14568E
	C	U14566E
	Project manager	U14569E
	Assembly language	U14567E
ID850 (Ver.2.20) (Integrated debugger)	Operation Windows-based	U14580E
SM850 (Ver.2.20) (System simulator)	Operation Windows-based	U14782E
RX850 (Real-time OS)	Basics	This manual
	Installation	U13410E
	Technical	U13431E
RX850 Pro (Real-time OS)	Basics	U13773E
	Installation	U13774E
	Technical	U13772E
RD850 (Task debugger)		U13737E
RD850 Pro (Task debugger)		U13916E
AZ850 (System performance analyzer)		U14410E
PG-FP3 (Flash memory programmer)		U13502E

TABLE OF CONTENTS

LIST OF FIGURES (1/2)

LIST OF FIGURES (2/2)

LIST OF TABLES

[MEMO]

CHAPTER 1   OVERVIEW

1.1 OVERVIEW

The RX850 is a built-in real-time, multitasking control OS that provides a highly efficient real-time, multitasking environment to increases the application range of the V850 family control units.

The RX850 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 to 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 MULTITASKING 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 multitasking OS enables the parallel processing of tasks by switching the tasks to be executed as determined by the system.

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

1.4 FEATURES

The RX850 has the following features:

(1) Conformity with μITRON 3.0 specification

The RX850 is designed to conform with the μITRON 3.0 specification, that defines a typical built-in control OS architecture. The RX850 implements μITRON 3.0 functions of up to level S.
The μITRON 3.0 specification applies to a built-in, real-time control OS.

(2) High generality

The RX850 supports all the system calls specified by the μITRON 3.0 specification to offer superior application system generality.
The RX850 can be used to create a real-time, multitasking 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.

(3) Realization of real-time processing and multitasking

The RX850 supports the following functions to realize complete real-time processing and multitasking:

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

(4) Scheduling lock function

The RX850 supports functions that allow a user processing program to disable and resume dispatching (task scheduling).

(5) Compact design

The RX850 is a real-time, multitasking 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.

(6) Utilization of original instructions

The high-speed execution speed of the V850 family, combined with the original instructions, enables high-speed processing.

(7) Utility support

The RX850 supports the following utility to aid in system construction:

- CF850 (configurater)

1.5 CONFIGURATION

The RX850 consists of three subsystems: the nucleus, system initialization, and cofigurater.
These subsystems are outlined below:

(1) Nucleus

The nucleus forms the heart of the RX850, a system that supports real-time, multitasking control. The nucleus provides the following functions:

- Generation/initialization of a management object
- Processing of a system call issued by a program being processed (task, handler)
- Selection of the program (task, handler) 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.

(2) System initialization

The system initialization supports both hardware initialization, necessary to enable the operation of the RX850, and software initialization.
Upon the activation of a system running under the RX850, system initialization is executed first.
The RX850 provides sample source files for processing that is dependent on the hardware configuration of the execution environment (reset routine), as well as for the processing for customizing the user software environment (initialization handler).
This improves portability to target systems and facilitates customization.

(3) Configurater CF850

To construct a system using the RX850, information files containing the data to be supplied to the RX850 (system information 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. But, files written in such a way are relatively difficult to write and subsequently understand.
The RX850 provides Configurater CF850, a utility tool for inputting data in interactive mode and outputting the results as an information file.
Configurater CF850 allows the user to easily create a new information file or change an existing information file.

1.6 APPLICATIONS

The RX850 can be applied to the following systems:

- Control over systems which use servo motors

Example	PPCs
	Printers
	NC machine tools

- Control over systems which require a rapid response

Example	Engines
	Cellular telephones
	PHS
	Digital still cameras

★ 1.7 EXECUTION ENVIRONMENT

The RX850 has been developed as an OS for embedded control and runs on a target system equipped with the following hardware.

(1) Target CPU

- V851
- V852
- V853
- V854
- V850/SA1
- V850/SBx
- V850/SV1
- V850E/MS1
- V850E/MA1
- NB85E core
- V850E/IA1

(2) Peripheral controller

The RX850 eliminates the hardware-dependent portions from the nucleus and supplies them as sample source files, in order to support a range of execution environments. If these sample source files are rewritten for the respective target systems, a specific peripheral controller is not required.

★ ## 1.8 DEVELOPMENT ENVIRONMENT

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

1.8.1 Hardware Environment

(1) Host machine

- PC-9800 series
- PC/AT™-compatible machine
- SPARCstation™
- HP9000 series 700™

(2) In-circuit emulators

- IE-703002-MC (V851, V852, V853, V854, V850/SA1, V850/SBx, V850/SV1)
- IE-703102-MC (V850E/MS1)
- IE-V850E-MC-A (V850E/MA1, NB85E core)
- IE-V850E-MC (V850E/IA1)

(3) I/O board for in-circuit emulator

- IE-703003-MC-EM1 (V853)
- IE-703008-MC-EM1 (V854)
- IE-703017-MC-EM1 (V850/SA1)
- IE-703037-MC-EM1 (V850/SBx)
- IE-703040-MC-EM1 (V850/SV1)
- IE-703102-MC-EM1 (V850E/MS1 5 V)
- IE-703102-MC-EM1-A (V850E/MS1 3.3 V)
- IE-703107-MC-EM1 (V850E/MA1)
- IE-703116-MC-EM1 (V850E/IA1)
- IE-V850E-MC-EM1-A (NB85E core 5 V)
- IE-V850E-MC-EM1-B (NB85E core 3.3 V)

Caution These I/O boards must be used in combination with the in-circuit emulator.

(4) PC interface boards

- IE-70000-98-IF-C (for PC-9800 series C bus)
- IE-70000-PC-IF-C (for PC/AT-compatible machines ISA bus)
- IE-70000-CD-IF-A (for PCMCIA socket)
- IE-70000-PCI-IF (for PCI bus)

1.8.2 Software Environment

(1) OS (( ): host machine)

- Windows 95/Windows 98/Windows NT™ 4.0 (PC-9800 series, PC/AT-compatible machines)
- Solaris™ 2.x (SPARCstation)
- SunOS™ 4.1.x (SPARCstation)

(2) Cross tools

- CA850 (NEC Corporation)
- CCV850 (Green Hills Software Inc.)

(3) Debuggers

- ID850 (NEC Corporation)
- SM850 (NEC Corporation)
- MULTI™ (Green Hills Software Inc.)
- PARTNER (Kyoto Microcomputer)

(4) Task debugger

- RD850 (NEC Corporation)

(5) System performance analyzer

- AZ850 (NEC Corporation)

1.9 SYSTEM CONSTRUCTION PROCEDURE

System construction involves incorporating created load modules into a target system, using the file group copied from the RX850 distribution media to the user development environment (host machine).

The system construction procedure is outlined below.

For details, refer to the **RX850 User's Manual, Installation (U13410E)**.

(1) Creating an information file

- System information table
- System information header file

(2) Creating system initialization

- Reset routine
- Initialization handler

(3) Creating idle handlers

(4) Creating processing programs

- Task
- Directly activated interrupt handler
- Indirectly activated interrupt handler
- Cyclic handler

(5) Creating an initialization data save area

(6) Creating a link directive file

(7) Creating a load module

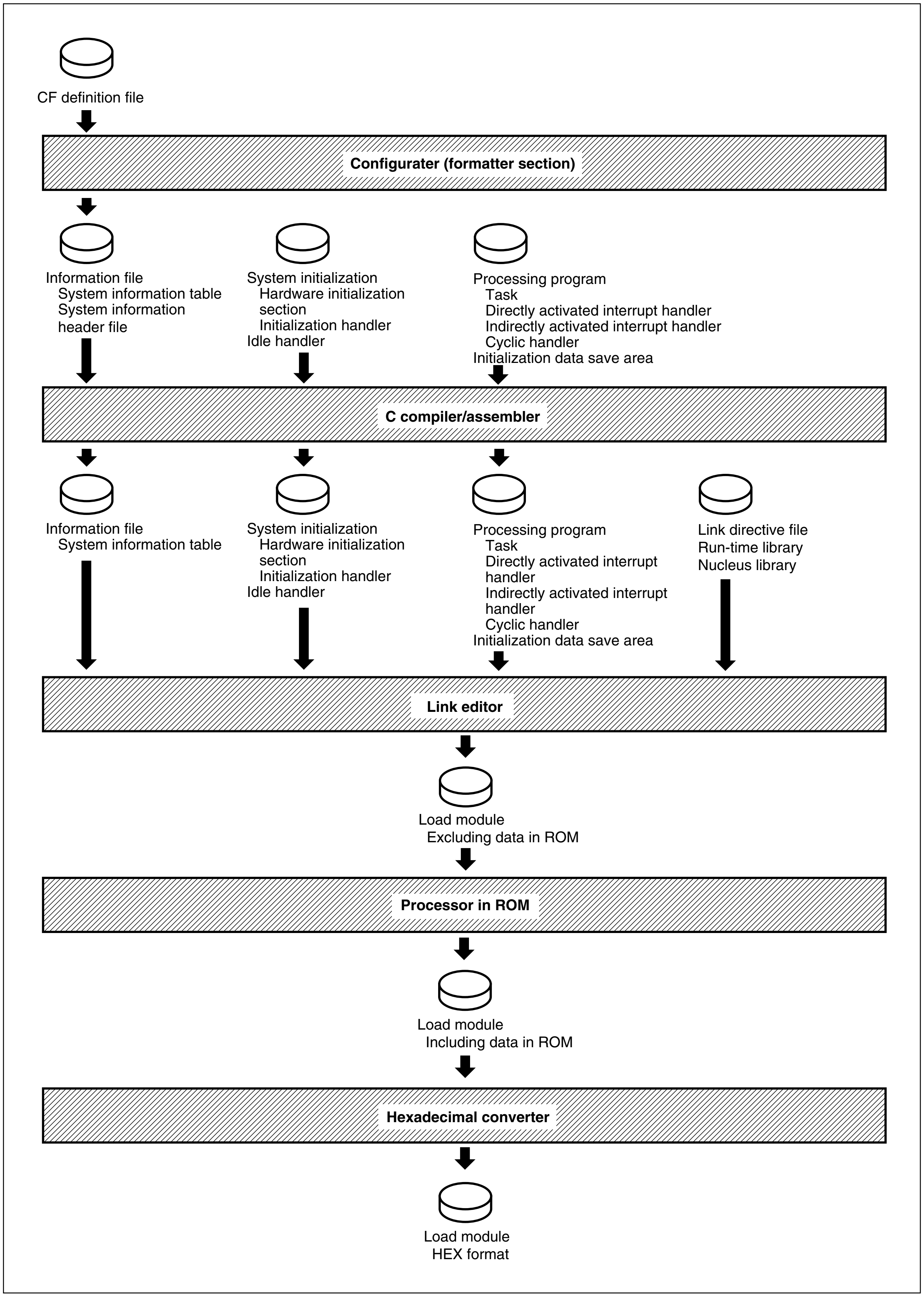
(8) Incorporating the load module into the system

Caution When the CCV850 is used, an initialization data save area need not be created.

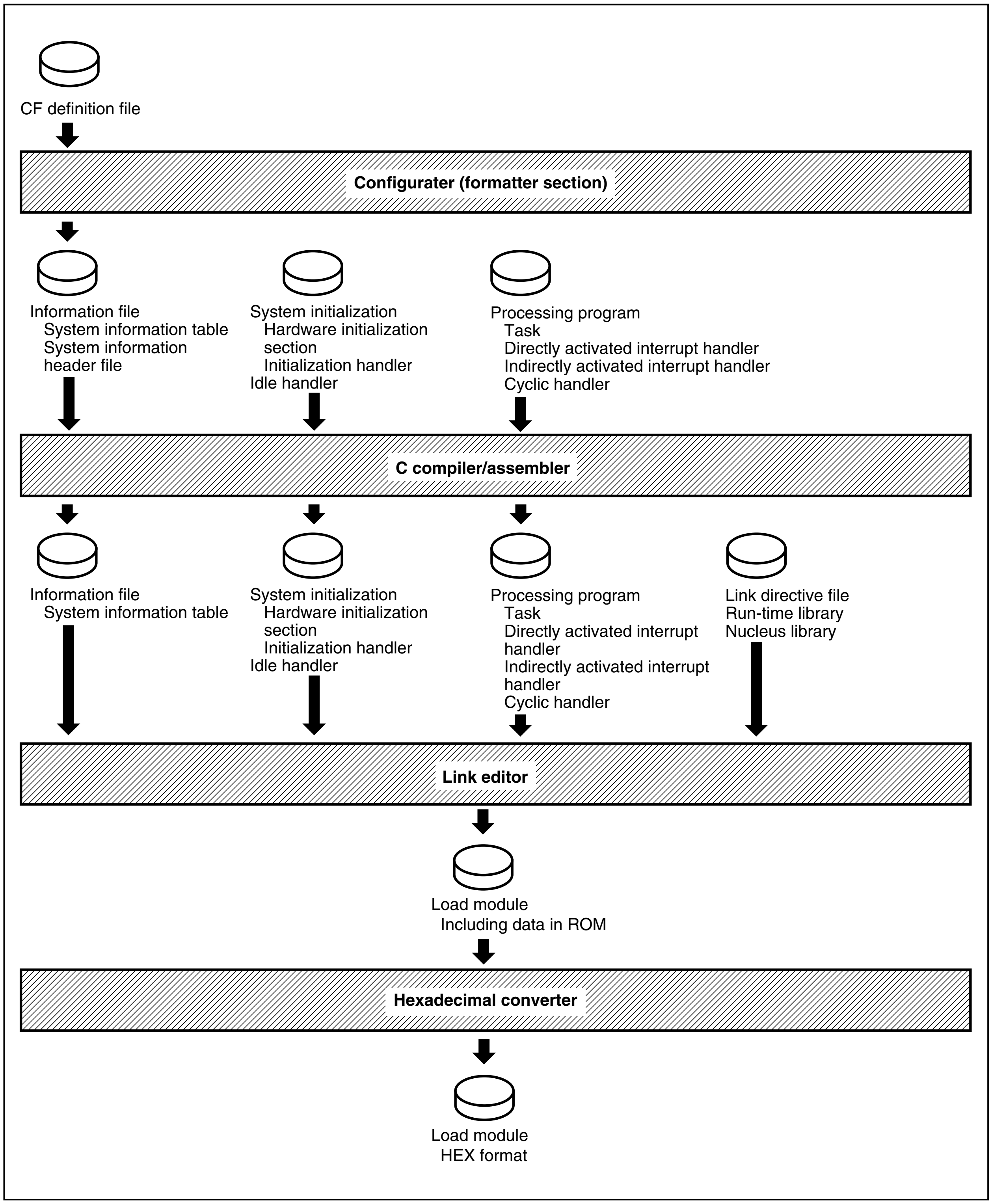
Figures 1-1 and 1-2 show the system construction procedures.

Figure 1-1 illustrates the system construction procedure when the CA850 is used. Figure 1-2 illustrates the system construction procedure when the CCV850 is used.

★ Figure 1-1. System Construction Procedure When Using CA850



★ **Figure 1-2. System Construction Procedure When Using CCV850**



[MEMO]

CHAPTER 2 NUCLEUS

This chapter describes the nucleus, the heart of the RX850.

2.1 OVERVIEW

The nucleus forms the heart of the RX850, a system that supports real-time, multitasking control. The nucleus provides the following functions:

- Initialization of a management object
- Processing of a system call issued by a program being processed (task, handler)
- Selection of the program (task, handler) to be executed next, according to an event that occurs internal to or external to the target system

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

2.2 FUNCTIONS

The nucleus consists of management modules and a scheduler.
This section overviews the functions of the management modules and scheduler.
See **Chapters 3** to **8** 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 the RX850. For example, the module can start and terminate a task.

(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, 1-bit event flag
• Communication function	:	Mailbox

(3) Interrupt management function
This module performs the processing related to an interrupt, such as the activation of an interrupt handler, return from an interrupt handler, disablement or resumption of acceptance of maskable interrupts, and change or acquisition of the contents of an interrupt control register.

(4) Memory pool management function

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

(a) RX850 area

- Management objects
- Memory pool

(b) Processing program (task, handler) area

- Text area
- Data area
- Stack area

The RX850 also applies dynamic memory pool management. For example, the RX850 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 task timeout wait or activation of a cyclic handler) that is based on clock interrupts generated by the hardware at regular intervals.

(6) Scheduler

This module manages and determines the order in which tasks are executed and manages how the processor is applied to individual tasks.
The RX850 determines the task execution order according to assigned priority method 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 assigns the task processing time.

Caution In the RX850, the smaller the value of the priority assigned to the task, the higher the priority.

CHAPTER 3   TASK MANAGEMENT FUNCTION

This chapter describes the task management performed by the RX850.

3.1 OVERVIEW

Tasks are execution entities of arbitrary sizes, such that they are difficult to manage directly. The RX850 manages task states and tasks themselves by using management objects that correspond to tasks on a one-to-one basis.

3.2 TASK EXECUTION RIGHT

Each task requires a "stack." The stack for a task (task stack) is used to store information such as the register information that is used when a task is switched and information on the values of the automatic variables used in a task. Because a task stack must be allocated to each task, the amount of RAM consumed for the stack increases considerably.

In a system, however, some tasks are never executed at the same time. If these tasks can share a stack, the amount of RAM consumed can be reduced substantially.

The RX850 groups tasks and provides related functions. One of the functions **allows only one task to be started in a single group**. The task grouping functions allow the memory areas allocated to tasks by the RX850 (task stack areas) to be shared, thus assisting the user in optimizing the efficiency of memory use. A group is called a **task execution right group**. A **task execution right** is the processing right assigned when a task is started. In other words, a task having the task execution right in a group is ready to be executed. To execute other tasks in the same group, the task currently having the execution right must relinquish the right. This is done by using system call `ext_tsk` (terminates the task issuing this system call) or `ter_tsk` (terminating the other system call).

Cautions 1. If a task cannot acquire the execution right after it has been activated, it enters the task execution right wait state.

2. 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. The task context is allocated in the task stack.

3.3 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.

The RX850 classifies task states into the following six types:

(1) `Dormant` state

A task in this state is not started or has already completed its processing.

A task in the `dormant` state is not scheduled by the RX850.

This state differs from the `wait` state in the following points:

- All resources are released.
- The execution environment information (task context) given by the program counter, work registers, and the like is initialized when the processing is resumed.
- A state manipulation system call (`ter_tsk`, `chg_pri`, `sus_tsk`, `wup_tsk`, etc.) causes an error.

(2) `Ready` state

A task in this state has the task execution right and is ready to perform its processing. This task waits for a processing time to be assigned while another task having a higher (or the same) priority level is performing its processing.

A task in the `ready` state is scheduled by the RX850.

(3) `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.

(4) `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. Execution environment information (task context) given by the program counter, work registers, and the like that was being used up until the stop is restored.

The RX850 further divides tasks in the `wait` state into the following seven groups, according to the conditions:

(a) Task execution right wait state

A task enters this state if it cannot obtain the task execution right from the relevant task execution right group upon the issue of an `sta_tsk` system call.

(b) 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.

(c) 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.

(d) 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.

(e) 1-bit event flag wait state

A task enters this state if `1` is not set in a relevant 1-bit event flag upon the issue of a `vwai_flg1` or `vtwai_flg1` system call.

(f) 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.

(g) 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_blf`, `tget_blf`, `get_blk`, or `tget_blk` system call.

(h) Timeout wait state

A task enters this state upon the issue of a `dly_tsk` system call.

(5) `Suspend` state

A task in this state has been stopped by a system call issued by another task.

The processing of this task is resumed from the point at which it was stopped. Execution environment information (task context) given by the program counter, work registers, and the like that was being used up until the stop is restored.

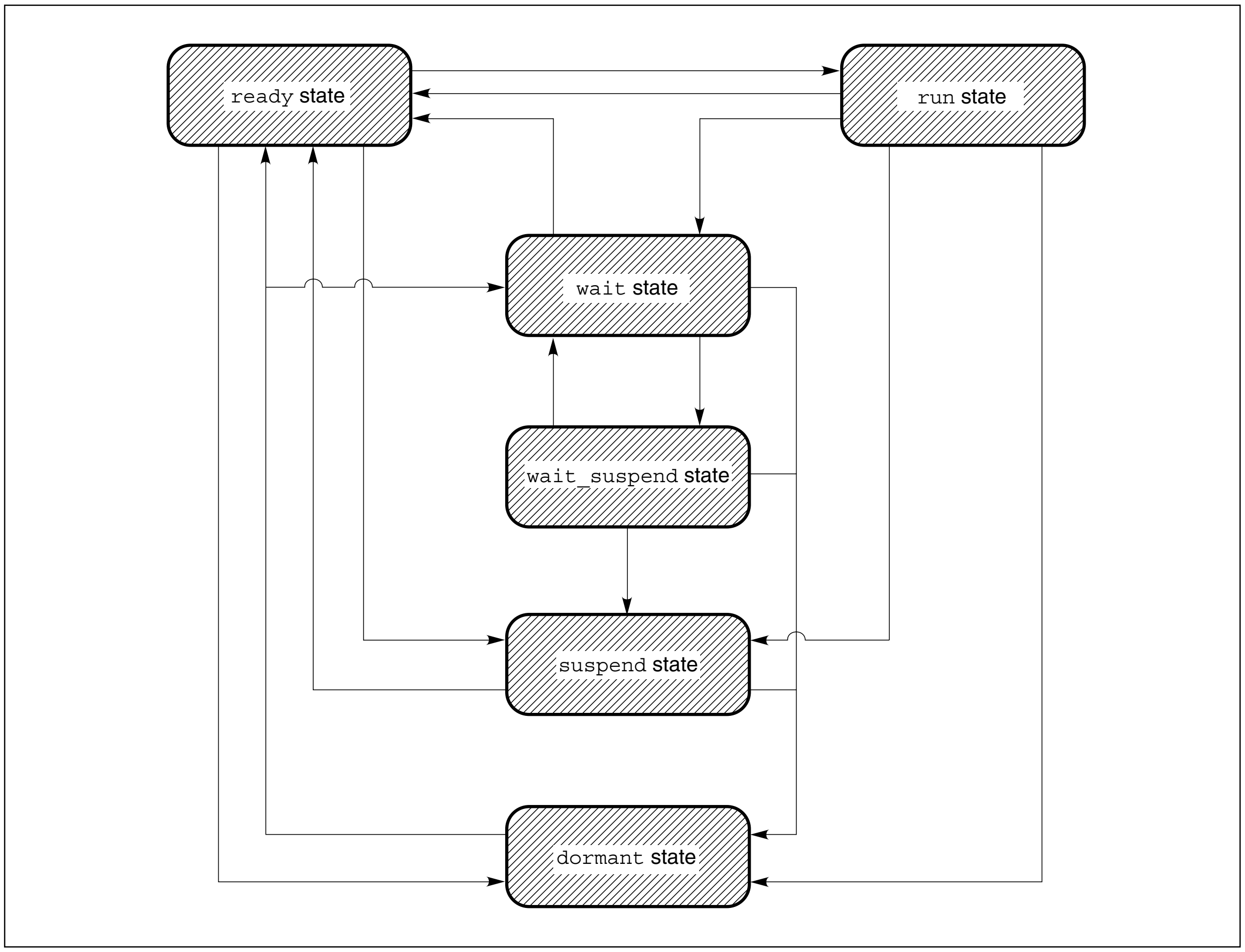
(6) `Wait_suspend` state

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

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

Figure 3-1 shows the relationship between task states.

Figure 3-1. Task State Transition



3.4 TASK GENERATION

RX850 tasks to be used by the system are specified at configuration. In other words, RX850 tasks can only be statically generated using the information specified at configuration (generation during system initialization). They cannot be dynamically generated using system calls.

To generate RX850 tasks, an area for managing each task (the management object) is allocated and initialized.

The RX850 recognizes such an area allocated in memory as a task and manages it accordingly.

The following task information is specified at configuration:

- Task name
- Task activation address
- Task stack size
- Type of system memory in which task stacks are allocated (`.pool0` or `.pool1` section)
- Initial task priority
- Initial task state
- Task activation code
- Specification of whether acceptance of maskable interrupts is enabled or disabled at the time of task activation

Caution For task system calls, an ID number is used to specify the task to be manipulated (target task). Configurater CF850 consecutively assigns integers, starting with `0x1` to tasks as ID numbers, based on task information stored in the system information table.

3.5 TASK ACTIVATION

Task activation under the RX850 involves assigning a task execution right to a task in the `dormant` state, and switching that task to the `ready` state.

A task is activated by issuing an `sta_tsk` system call.

- `sta_tsk` system call

  Issuing the `sta_tsk` system call assigns a task execution right to the task specified by the parameter, then switches that task from the `dormant` state to the `ready` state.

  When this system call is issued, if the task cannot acquire a task execution right from the relevant task execution right group, the task itself is queued at the end of the queue of this task execution right group. Thus, the task leaves the `dormant` state and enters the `wait` state (the task execution right wait state).

3.6 TASK TERMINATION

Task termination under the RX850 involves switching a task to `dormant` state and excluding that task from those to be scheduled by the RX850.

Under the RX850, 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: A task is terminated by another task when the task must immediately stop the processing because of an error that occurs prior to the completion of the processing.

The task terminates upon the issue of an `ext_tsk` or `ter_tsk` system call.

- `ext_tsk` system call
  A task which issued the `ext_tsk` system call is switched from the `run` state to the `dormant` state.
- `ter_tsk` system call
  A task specified by the parameters is forcibly switched to the `dormant` state.

3.7 IN-TASK PROCESSING

The RX850 utilizes a unique means of scheduling to switch tasks.

Note the following when coding task processing.

(1) Saving and restoring the contents of a register

When switching tasks, the RX850 saves and restores the contents of work registers in line with the function call conventions of the C compiler (CA850 or CCV850). This eliminates the need for coding processing to save the contents at the beginning of a task and that for restoring 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.

Caution When switching tasks, the RX850 does not switch the gp, tp, and ep registers.

(2) Switching stacks

When switching tasks, the RX850 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

The following system calls can be issued within a task:

- Task management system calls

```
sta_tsk     ext_tsk     ter_tsk     dis_dsp     ena_dsp
chg_pri     rot_rdq     rel_wai     get_tid     ref_tsk
```

- Task-associated synchronization system calls

```
sus_tsk     rsm_tsk     frsm_tsk    slp_tsk     tslp_tsk
wup_tsk     can_wup
```

- Synchronous communication system calls

```
sig_sem     wai_sem     preq_sem    twai_sem    ref_sem
set_flg     clr_flg     wai_flg     pol_flg     twai_flg
ref_flg     vset_flg1   vclr_flg1   vwai_flg1   vpol_flg1
vtwai_flg1  vref_flg1   snd_msg     rcv_msg     prcv_msg
trcv_msg    ref_mbx
```

- Interrupt management system calls

```
loc_cpu     unl_cpu     dis_int     ena_int     chg_icr
ref_icr
```

- Memory pool management system calls

```
get_blf     pget_blf    tget_blf    rel_blf     ref_mpf
get_blk     pget_blk    tget_blk    rel_blk     ref_mpl
```

- Time management system calls

```
dly_tsk     act_cyc     ref_cyc
```

- System management system calls

```
get_ver     ref_sys
```

3.8 ACQUIRING AN ID NUMBER

To acquire the ID number of a task being executed, use system call `get_tid`.

Because the ID number of a task is specified with a symbol by the configuration file when the task is created, that system is usually used as the ID number of the task. If, however, the same program code is shared by two or more tasks, this system call is used to determine which task is executing the program code.

3.9 ACQUIRING TASK INFORMATION

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

(1) `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

`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
`TTS_WTX`	`(H'20)`	: Task execution right wait state
`TTS_WTS`	`(H'28)`	: Task execution right wait + `suspend` state

- Type of the 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)`	: Variable-size memory block wait state
`TTW_MPF`	`(H'2000)`	: Fixed-size memory block wait state
`TTW_1FLG`	`(H'4000)`	: 1-bit event flag wait state

- ID number of the object to be processed

CHAPTER 4   SYNCHRONOUS COMMUNICATION FUNCTIONS

This chapter explains the manages synchronous communication functions performed by the RX850.

4.1 OVERVIEW

In an environment in which multiple tasks are executed concurrently (multitasking), the next task to be executed, or the contents of the processing performed by a task, may depend on the processing results output by a preceding task. In other words, a task may set the execution conditions for another's processing, or the contents of tasks' processing may be mutually related.

Therefore, functions enabling communication between such tasks are required. These functions are used when a task stops its processing to await the results output by another task, and when a task must wait until a condition required to continue processing is satisfied.

In the RX850, these functions are called **synchronization functions**. There are two synchronization functions: the exclusive control function and the wait function. The RX850 provides semaphores for the exclusive control function and event flags and 1-bit event flags for the wait function.

For multitasking, an inter-task communication function is also required to allow a task to receive the processing results output by another.

In the RX850, this function is called the **communication function**. The RX850 provides mailboxes for the communication function.

4.2 SEMAPHORES

In multitasking, a function is required to prevent resource contentions from occurring. A resource contention occurs when concurrently executing multiple tasks simultaneously use a limited number of resources such as A/D converters, coprocessors, files, and programs. To prevent contentions from occurring, the RX850 provides non-negative counter-type semaphores.

A semaphore is a counter used to manage the number of resources. An the RX850 semaphore is a 7-bit counter used to exercise exclusive control over tasks.

The following semaphore system calls are used to dynamically operate 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.

Cautions 1. For the RX850, the elements required to execute a task are called resources. In other words, resources include all hardware components such as A/D converters and coprocessors, as well as software components such as files and programs.

2. For semaphore system calls, an ID number is used to specify the semaphore to be manipulated (target semaphore).
Configurater CF850 consecutively assigns integers, starting with `0x1`, to semaphores as ID numbers, based on semaphore information stored in the system information table.

4.2.1 Semaphore Generation

RX850 semaphores to be used by the system are specified at configuration. In other words, RX850 semaphores can only be statically generated using the information specified at configuration (generation during system initialization). They cannot be dynamically generated using system calls.

To generate RX850 semaphores, an area for managing each semaphore (the management object) is allocated and initialized.

The RX850 recognizes such an area allocated in memory as a semaphore and manages it accordingly.

The following semaphore information is specified at configuration:

- Semaphore name
- Initial count of semaphore resources

4.2.2 Returning a Resource

A resource is returned by issuing a `sig_sem` system call.

(1) `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.

4.2.3 Acquiring Resources

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

(1) `wai_sem` system call

By issuing the `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 placed at the end of the queue of this semaphore. Thus, the task leaves the `run` state and enters the `wait` state (the resource wait state).

The task shall be released from this resource wait state and return to the `ready` state in the following cases:

- When a `sig_sem` system call is issued.
- When a `rel_wai` system call is issued and the wait state is forcibly canceled.

Caution Tasks are queued into the queue of the specified semaphore in the order (FIFO) in which the tasks make resource acquisition requests.

(2) `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.

(3) `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 placed at the end of the queue of this semaphore. Thus, the task leaves the `run` state and enters the `wait` state (the resource wait state).

The task shall be released from this resource wait state and returned to the `ready` state in the following cases:

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

Caution Tasks are queued into the queue of the specified semaphore in the order (FIFO) in which the tasks make resource acquisition requests.

4.2.4 Acquiring Semaphore Information

Semaphore information is acquired by issuing the `ref_sem` system call.

(1) `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

`FALSE(0)`	: No task is queued.
Value	: ID number of the first task in the queue

- The number of currently available resources

4.2.5 Exclusive Control Using Semaphores

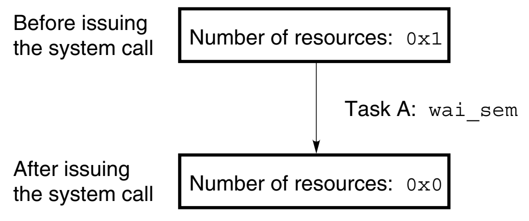
The following is an example of using semaphores to manipulate the tasks under exclusive control.

(Prerequisites)

- Task priority
  Task A > Task B
- State of tasks
  Task A: `run` state
  Task B: `ready` state
- Semaphore attribute
  Number of resources initially assigned to the semaphore: `0x1`

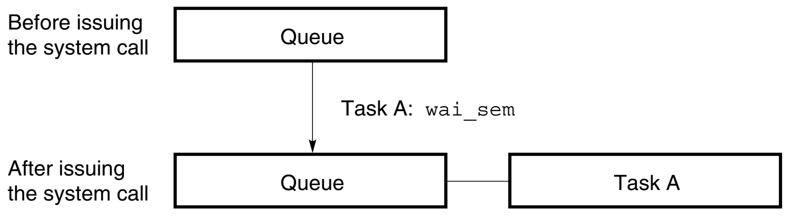
(1) Task A issues the `wai_sem` system call.
The number of resources assigned to this semaphore and managed by the RX850 is `0x1`. Thus, the RX850 decrements the semaphore counter by `0x1`.
At this time, task A does not enter the `wait` state (the resource wait state). Instead, it remains in the `run` state.
Figure 4-1 shows the state of the relevant semaphore counter.

Figure 4-1. State of Semaphore Counter



(2) Task A issues the `wai_sem` system call.
The number of resources assigned to this semaphore and managed by the RX850 is `0x0`. Thus, the RX850 changes the state of task A from `run` to the resource wait state and places the task at the end of the queue for this semaphore.
Figure 4-2 shows the state of the queue of this semaphore.

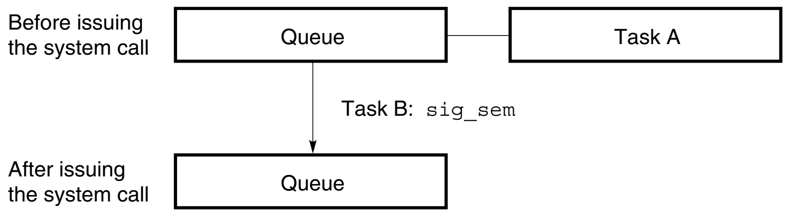
Figure 4-2. State of Queue



(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 the `ready` state.
Figure 4-3 shows the state of the queue of this semaphore.

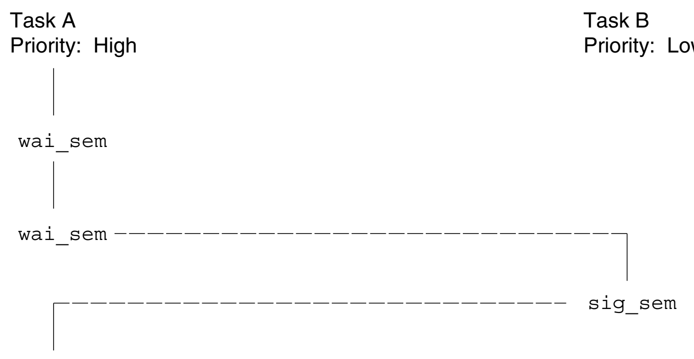
Figure 4-3. State of Queue



(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

For multitasking, an inter-task wait function is required to place a task in the wait state until another task outputs its processing results. The waiting task need only monitor the occurrence of an event upon which the processing results are output. To enable this, the RX850 provides event flags.

An event flag is aggregate data consisting of 1-bit flags indicating whether a particular event has occurred. For an RX850 event flag, 32 bits are handled as a set of information. A meaning can be assigned not only to each of the 32 bits but also to combinations of several bits.

The following event flag system calls are used to dynamically manipulate 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.

Caution For event flag system calls, an ID number is used to specify the event flag to be manipulated (target event flag).
Configurater CF850 consecutively assigns integers, starting from `0x1`, to event flags as ID numbers, based on event flag information stored in the system information table.

4.3.1 Event Flag Generation

RX850 event flags to be used by the system are specified at configuration. In other words, RX850 event flags can only be statically generated using information specified at configuration (generation at system initialization). They cannot be dynamically generated using system calls.

To generate RX850 event flags, an area for managing each event flag (the management object) is allocated and initialized.

The RX850 recognizes such an area allocated in memory as an event flag and manages it accordingly.

The following event flag information is specified at configuration:

- Event flag name

Cautions 1. If an event flag is generated during system initialization, the RX850 sets `0x0` as the initial bit pattern.
2. The RX850 specifies that only one task can be queued into the queue of an event flag.

4.3.2 Setting a Bit Pattern

A bit pattern is set by issuing a `set_flg` system call.

(1) `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.

4.3.3 Clearing a Bit Pattern

A bit pattern is cleared by issuing a `clr_flg` system call.

(1) `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 particularly careful attention to this point.

4.3.4 Checking Bit Patterns

Bit pattern check is performed by issuing the `wai_flg`, `pol_flg`, or `twai_flg` system call.

(1) `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 for the event flag specified by this system call parameter, the task that issues this system call is queued into the queue of this event flag. Thus, the task leaves the `run` state and enters the `wait` state (the event flag wait state).

The task shall be released from this event flag wait state, then return to the `ready` state, in the following cases:

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

(2) `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.

(3) `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 into the queue for this event flag. Thus, the task leaves the `run` state and enters the `wait` state (the event flag wait state).

The task shall be released from the event flag wait state and return to the `ready` state in the following cases:

- 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 `rel_wai` system call is issued and the event flag wait state is forcibly canceled.

Remark The RX850 allows the specification of either of the two wait conditions, as well as the processing to be performed when the specified wait condition is satisfied, as described below:

(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

- The bit pattern is cleared.

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

4.3.5 Acquiring Event Flag Information

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

(1) `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

`FALSE(0)`	:	No task is queued.
Value	:	ID number of the task in the queue

- Current bit pattern

4.3.6 Wait Function Using Event Flags

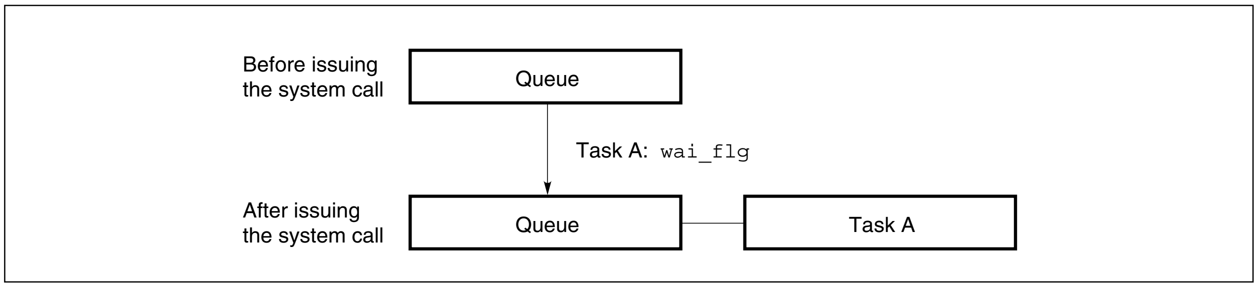
The following is an example of manipulating the tasks under wait and control using event flags.

(Prerequisites)

- Task priority
  Task A > Task B
- State of tasks
  Task A: `run` state
  Task B: `ready` state
- Event flag attribute
  Initial bit pattern: `0x0`

(1) Task A issues the `wai_flg` system call. The required bit pattern is `0x1`. The wait condition and the processing to be performed when the condition is satisfied are specified as `TWF_ANDW|TWF_CLR`.
The current bit pattern of the relevant event flag managed by RX850 is `0x0`. Thus, RX850 changes the state of task A from `run` to `wait` (the event flag wait state). Then, task A is queued into the queue for this event flag.
Figure 4-5 shows the state of the queue of this event flag.

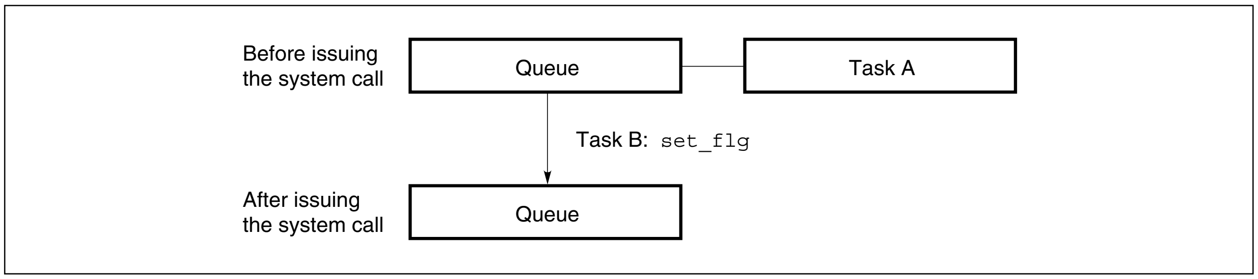
Figure 4-5. State of Queue



(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_flg` system call, the bit pattern of this event flag is cleared.
Figure 4-6 shows the state of the queue for this event flag.

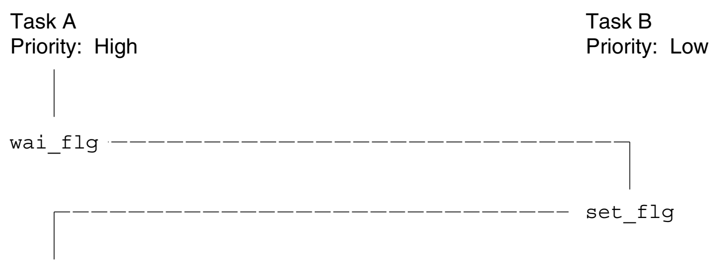
Figure 4-6. State of Queue



(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 1-BIT EVENT FLAGS

For multitasking, an inter-task wait function is required to place a task in the wait state until another task outputs its processing results. The waiting task need only monitor the occurrence of an event upon which the processing results are output. To enable this, the RX850 provides 1-bit event flags.

A 1-bit event flag is a single bit indicating whether a particular event has occurred. For an RX850 1-bit event flag, one bit is handled as a unit of information.

Only one task can be queued for one event flag if an event flag is used as described in **Section 4.3**. However, this 1-bit event flag can queue two or more tasks. It is also used to simultaneously release two or more tasks from the wait state.

The following 1-bit event flag system calls are used to dynamically manipulate a 1-bit event flag:

`vset_flg1`	:	Sets a bit.
`vclr_flg1`	:	Clears a bit.
`vwai_flg1`	:	Checks a bit.
`vpol_flg1`	:	Checks a bit (by polling).
`vtwai_flg1`	:	Checks a bit (with timeout setting).
`vref_flg1`	:	Acquires 1-bit event flag information.

Caution For 1-bit event flag system calls, an ID number is used to specify the 1-bit event flag to be manipulated (target 1-bit event flag).
Configurater CF850 consecutively assigns integers, starting from `0x1`, to 1-bit event flags as ID numbers, based on 1-bit event flag information stored in the system information table.

4.4.1 1-Bit Event Flag Generation

RX850 1-bit event flags to be used by the system are specified at configuration. In other words, RX850 1-bit event flags can only be statically generated using information specified at configuration (generation at system initialization). They cannot be dynamically generated using system calls.

To generate RX850 1-bit event flags, an area for managing each 1-bit event flag (the management object) is allocated and initialized.

The RX850 recognizes such an area allocated in memory as a 1-bit event flag and manages it accordingly.

The following 1-bit event flag information is specified at configuration:

- 1-bit event flag name

Caution When a 1-bit event flag is generated during system initialization, the RX850 sets `0` as the initial bit value.

4.4.2 Setting a Bit

A bit is set by issuing a `vset_flg1` system call.

(1) `vset_flg1` system call

The `vset_flg1` system call sets `1` in the 1-bit event flag specified by a parameter.

When this system call is issued, if any tasks have been queued into the queue of the specified 1-bit event flag, the first task to the task which specifies bit clearing are removed from the queue.

Then, these task will leave the `wait` state (the 1-bit event flag wait state) and enter the `ready` state. Or, they will leave the `wait_suspend` state and enter the `suspend` state.

4.4.3 Clearing a Bit

A bit is cleared by issuing a `vclr_flg1` system call.

(1) `vclr_flg1` system call

The `vclr_flg1` system call sets 0 in the 1-bit event flag specified by a parameter.

This system call does not queue clear requests. Note that, if the 1-bit event flag specified in the current `vclr_flg1` system call has already been cleared by the previous `vclr_flg1` system call, no processing is performed and it is not handled as an error.

4.4.4 Checking a Bit

Bit check is performed by issuing the `vwai_flg1`, `vpol_flg1`, or `vtwai_flg1` system call.

(1) `vwai_flg1` system call

The `vwai_flg1` system call checks whether `1` is set in the 1-bit event flag specified by a parameter.

When this system call is issued, if `1` is not set in the specified 1-bit event flag, the task that issued this system call is queued at the end of the queue of this 1-bit event flag. Thus, the task leaves the `run` state and enters the `wait` state (the 1-bit event flag wait state).

The task shall be released from this 1-bit event flag wait state, then return to the `ready` state, in the following cases:

- When a `vset_flg1` system call is issued.
- When a `rel_wai` system call is issued and the 1-bit event flag wait state is forcibly canceled.

Caution Tasks are queued into the queue of the specified 1-bit event flag in the order (FIFO) in which bit checking for the tasks was performed.

(2) `vpol_flg1` system call

The `vpol_flg1` system call checks whether `1` is set in the 1-bit event flag specified by a parameter.

When this system call is issued, if `1` is not set in the specified 1-bit event flag, `E_TMOUT` is returned as the return value.

(3) `vtwai_flg1` system call

The `vtwai_flg1` system call checks whether `1` is set in the 1-bit event flag specified by a parameter.
When this system call is issued, if `1` is not set in the specified 1-bit event flag, the task that issued this system call is queued at the end of the queue of this 1-bit event flag. Thus, the task leaves the `run` state and enters the `wait` state (the 1-bit event flag wait state).
The task shall be released from the 1-bit event flag wait state and return to the `ready` state in the following cases:

- Once the given wait time specified by parameter has elapsed.
- When a `vset_flg1` system call is issued.
- When a `rel_wai` system call is issued and the 1-bit event flag wait state is forcibly canceled.

Caution Tasks are queued into the queue of the specified 1-bit event flag in the order (FIFO) in which bit checking for the tasks was performed.

Remark RX850 allows the specification of the processing to be performed when the wait condition specified for the 1-bit event flag is satisfied, as follows:

(1) When the condition is satisfied

- The bit is not cleared.
  When the wait condition specified for the 1-bit event flag is satisfied, the bit of the 1-bit event flag is not cleared.
- The bit is cleared.
  When the wait condition specified for the 1-bit event flag is satisfied, the bit of the 1-bit event flag is cleared.

4.4.5 Acquiring 1-Bit Event Flag Information

1-bit event flag information is acquired by issuing the `vref_flg1` system call.

- `vref_flg1` system call
  By issuing the `vref_flg1` system call, the task acquires the 1-bit event flag information (extended information, queued tasks, etc.) for the 1-bit event flag specified by a parameter.
  Details of 1-bit event flag information are as follows:

- Extended information
- Whether tasks are queued

`FALSE(0)`	:	No task is queued.
Value	:	ID number of the first task in the queue

- Current bit value

4.4.6 Wait Function Using 1-Bit Event Flags

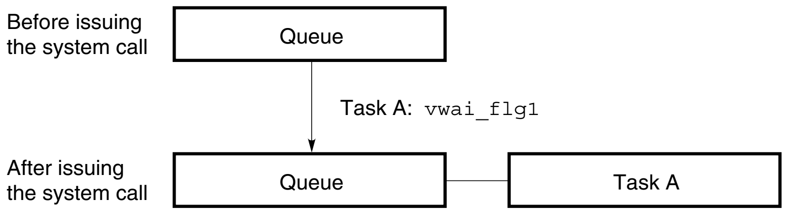
The following is an example of manipulating the tasks under wait and control using 1-bit event flags.

(Prerequisites)

- Task priority
  Task A > Task B
- State of tasks
  Task A: `run` state
  Task B: `ready` state
- 1-bit event flag attribute
  Initial bit value: `0`

(1) Task A issues the `vwai_flg1` system call. The processing performed when the condition is satisfied is specified as `TWF_CLR`.
The current bit pattern of the relevant 1-bit event flag managed by the RX850 is `0`. Thus, the RX850 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 1-bit event flag.
Figure 4-8 shows the state of the queue of this 1-bit event flag.

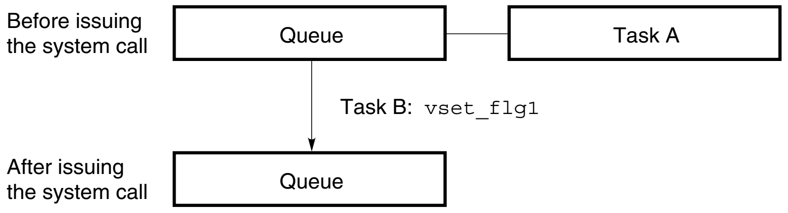
Figure 4-8. State of the Queue



(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 `vset_flg1` system call.
Task A that has been queued into the queue of the relevant 1-bit event queue leaves the event flag wait state and enters the `ready` state.
Since `TWF_CLR` was specified when task A issued the `vwai_flg1` system call, the bit pattern of this 1-bit event flag is cleared.
Figure 4-9 shows the state of the queue for this 1-bit event flag.

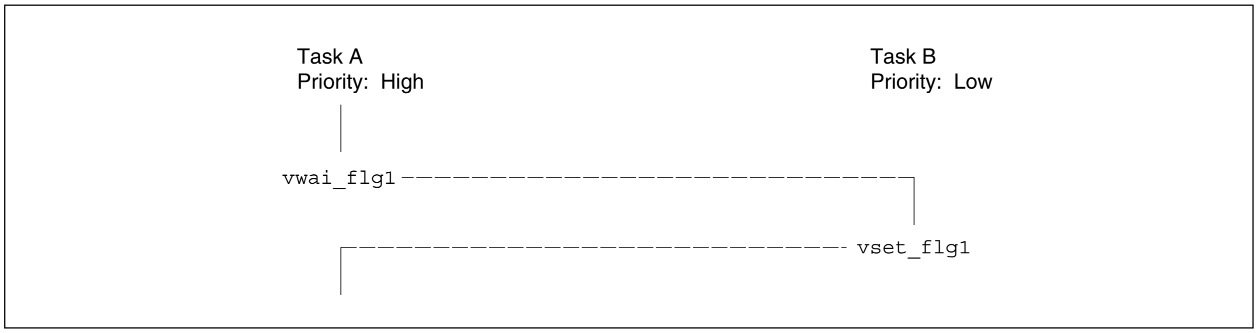
Figure 4-9. State of the Queue



(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 4-10 shows the transition of wait and control by 1-bit event flags in steps (1) to (4).

Figure 4-10. Wait and Control by 1-Bit Event Flags



4.5 MAILBOXES

In multitasking, an inter-task communication function is required to enable the results of a task's processing to be passed to other tasks. To enable this, the RX850 provides mailboxes.

RX850 mailboxes have one wait queue dedicated to tasks (task wait queue) and another dedicated to messages (message wait queue). They can be used with the inter-task wait function as well as with the inter-task message communication function.

The following mailbox system calls are used to dynamically manipulate 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.

Caution For mailbox system calls, an ID number is used to specify the mailbox to be manipulated (target mailbox).
Configurater CF850 consecutively assigns integers, starting with `0x1`, to mailboxes as ID numbers, based on the mailbox information stored in the system information table.

4.5.1 Mailbox Generation

RX850 mailboxes to be used by the system are specified at configuration. In other words, RX850 mailboxes can only be statically generated using the information specified at configuration (generation at system initialization). They cannot be dynamically generated using system calls.

To generate RX850 mailboxes, an area for managing each mailbox (the management object) is allocated and initialized.

The RX850 recognizes such an area allocated in memory as a mailbox and manages it accordingly.

The following mailbox information is specified at configuration:

- Mailbox name
- Method of queuing messages

4.5.2 Sending a Message

A message is sent by issuing a `snd_msg` system call.

(1) `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.

Caution Queuing of a message into the message 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 (during configuration).

4.5.3 Receiving a Message

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

(1) `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 task shall be released from this message wait state and return to the `ready` state in the following cases:

- When a `snd_msg` system call is issued.
- When a `rel_wai` system call is issued and the message wait state is forcibly canceled.

Caution Tasks are queued into the queue of the specified mailbox in the order (FIFO) in which the tasks make message reception requests. The tasks cannot be queued according to their priorities.

(2) `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.

(3) `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 task shall be released from this message wait state and return to the `ready` state in the following cases:

- When the given time specified by parameter has elapsed.
- When a `snd_msg` system call is issued.
- When a `rel_wai` system call is issued and the message wait state is forcibly canceled.

Caution Tasks are queued into the queue of the specified mailbox in the order (FIFO) in which the tasks make message reception requests.

4.5.4 Messages

Under the RX850, 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 the RX850, 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. Pay particularly careful attention to this point.

(1) Allocating message areas

NEC recommends that the memory pool managed by the RX850 be allocated for messages. To make a memory pool area available for a message, the task should issue a `get_blf`, `pget_blf`, `tget_blf`, `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.

Caution If the `rel_blf` or `rel_blk` system call is issued while a message is placed in the message queue of the relevant mailbox, its operation will be unpredictable.

(2) Composing messages

The RX850 does not specially stipulate the length and contents of a message to be transmitted to a mailbox. If a priority is assigned to the first four bytes of a message and to the message, a one-byte reserved area is necessary.

The first four bytes of a message are used as a link area that is used to queue that message. The fifth byte is an area in which the priority of a message is stored. The message itself is stored in the subsequent area. To allocate the area for the message, these bytes must be taken into consideration.

Cautions 1. If no priority is assigned to a message, the fifth byte and those that follow can be used as the message itself.

2. If a priority is assigned to a message, theoretically, the sixth byte and those that follow constitute the message itself. Actually, however, the message starts from the eighth byte because of the area alignment of the compiler.

3. The priority of a message can be specified within a range of 0 to 31.

4. Set the first four bytes of a message to "`0x0`" to transmit the message to a mailbox before issuing system call `snd_msg`.

5. When using a mailbox for programming in C, "structure of message `T_MSG`" provided by the RX850 can be used. For details, refer to the description of system call `snd_msg` in Section 10.5.3.

4.5.5 Acquiring Mailbox Information

Mailbox information is acquired by issuing a `ref_mbx` system call.

(1) `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

`FALSE(0)`	: No task is queued.
Value	: ID number of the first task in the queue

- Whether messages are queued

`NADR(-1)`	: No message is queued.
Value	: Address of the first message in the queue

4.5.6 Inter-Task Communication Using Mailboxes

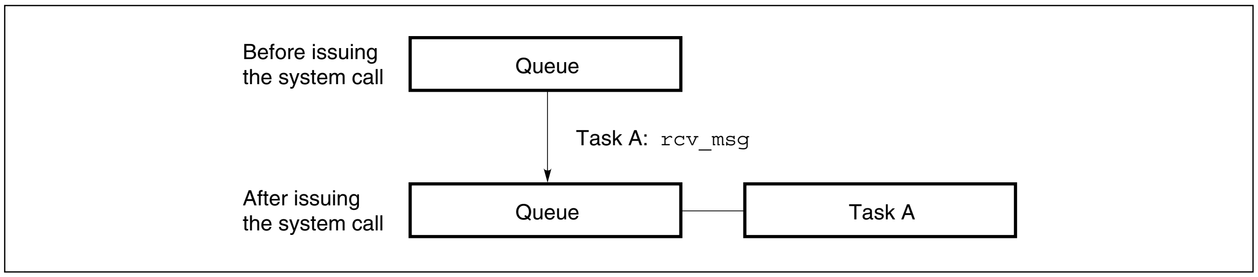
The following is an example of manipulating the tasks under inter-task communication using mailboxes.

(Prerequisites)

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

(1) Task A issues a `rcv_msg` system call.
No message is queued into the message queue of the relevant mailbox managed by the RX850. Thus, the RX850 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.
Figure 4-11 shows the state of the task queue for this mailbox.

Figure 4-11. State of Task Queue



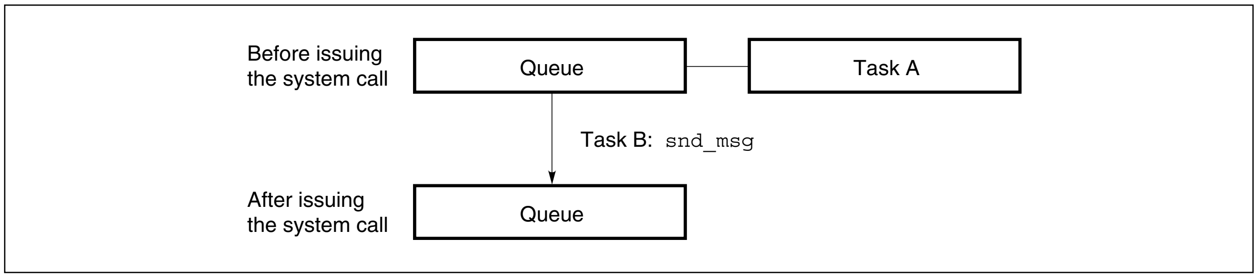
(2) As task A enters the message wait state, the state of task B changes from `ready` to `run`.

(3) Task B issues the `get_blf` 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.
Figure 4-12 shows the state of the task queue for this mailbox.

Figure 4-12. State of Task Queue

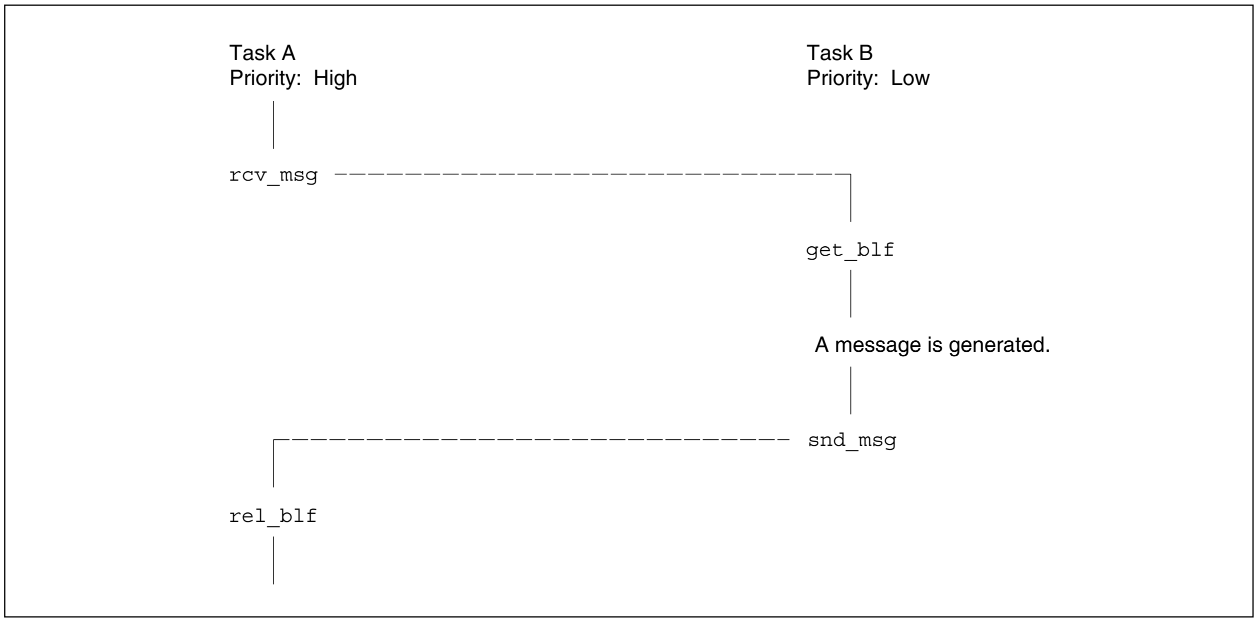


(6) The state of task A having the higher priority changes from `ready` to `run`.
At the same time, task B leaves the `run` state and enters the `ready` state.

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

Figure 4-13 shows the transition of inter-task communication in steps (1) to (7).

Figure 4-13. Inter-Task Communication Using Mailboxes



CHAPTER 5 INTERRUPT MANAGEMENT FUNCTION

This chapter describes the interrupt management function provided by the RX850.

5.1 OVERVIEW

The RX850 interrupt management function enables the following:

- Initiating an interrupt handler
- Return from an interrupt handler
- Disabling/resuming maskable interrupt acceptance
- Changing/acquiring the contents of an interrupt control register

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 RX850 provides two interfaces for interrupt handlers, enabling different levels of response, from the occurrence of an interrupt until the start of the interrupt handler.

- Directly activated interrupt handler
- Indirectly activated interrupt handler

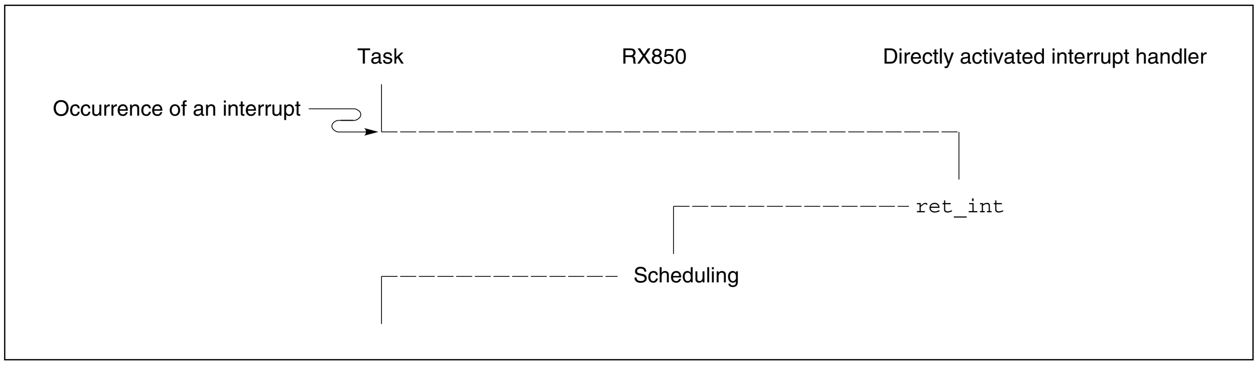
5.3 DIRECTLY ACTIVATED INTERRUPT HANDLER

The directly activated interrupt handler is a routine dedicated to interrupt processing, initiated without the intervention of the RX850 upon the occurrence of an interrupt.

Therefore, a rapid response that approaches the maximum level of hardware performance can be expected.

Figure 5-1 shows the flow of the processing performed by the directly activated interrupt handler.

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


Task
RX850
Directly activated interrupt handler
Occurrence of an interrupt
ret_int
Scheduling

5.3.1 Registering a Directly Activated Interrupt Handler

A directly activated interrupt handler may be registered by assigning it to an address to which control is passed by the processor upon the occurrence of an interrupt, or by writing and setting a branch instruction to a directly activated interrupt handler.

5.3.2 Internal Processing Performed by the Directly Activated Interrupt Handler

It is necessary to describe the internal processing performed by the directly activated interrupt handler, taking the following into account:

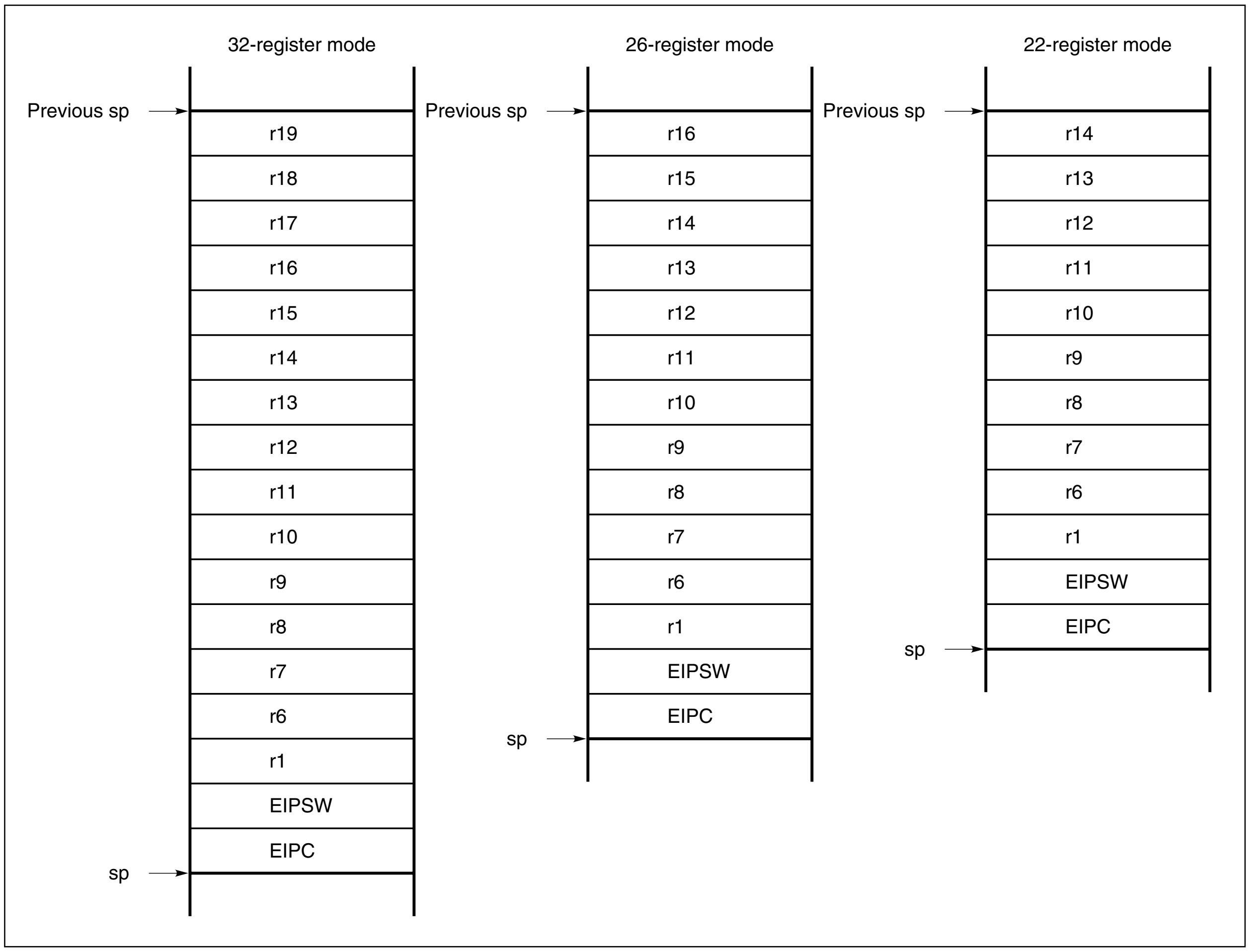
(1) Saving and restoring the contents of a register

When control is passed to a directly activated interrupt handler, the contents of the work registers remain as is upon the occurrence of the interrupt. If the work registers are used with the directly activated interrupt handler, the coding should be written such that they are saved when the interrupt handler starts, and restored upon the termination of interrupt handler operation.

This processing must be described in assembly language. To minimize the workload imposed on the user in describing processing in assembly language, the RX850 offers a macro for a directly activated interrupt handler. Because the registers are saved or restored in this macro, use this macro. For details of this macro, refer to the **RX850 User's Manual, Installation**.

Figure 5-2 shows the sequence in which the directly activated interrupt handler saves the contents of each register, for each register mode.

Figure 5-2. Saving the Register Contents



(2) Stack switching

When control is passed to a directly activated interrupt handler, the stack remains as is upon the occurrence of the interrupt. If the interrupt handler stack is used, the coding should be written such that the interrupt handler stack is switched at the start of the directly activated interrupt handler, returning to the original stack upon the completion of interrupt handler operation. If the macro for a directly activated interrupt handler is used, however, the processing for switching the stack does not have to be described because that processing is performed by the macro.

(3) Limitations imposed on system calls

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

- Task management system calls

```
sta_tsk     chg_pri     rot_rdq     rel_wai     get_tid
ref_tsk
```

- 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     vset_flg1   vclr_flg1   vpol_flg1
vref_flg1   snd_msg     prcv_msg    ref_mbx
```

- Interrupt management system calls

```
ret_int     ret_wup     dis_int     ena_int     chg_icr
ref_icr
```

- Memory pool management system calls

```
pget_blf    rel_blf     ref_mpf     pget_blk    rel_blk
ref_mpl
```

- Time management system calls

```
act_cyc     ref_cyc
```

- System management system calls

```
get_ver     ref_sys
```

(4) Return processing from the directly activated interrupt handler

Return processing from the directly activated interrupt handler is performed by issuing the `ret_int` or `ret_wup` system call upon the completion of interrupt handler operation.

- `ret_int` system call
  The `ret_int` system call performs return from the directly activated interrupt handler.
- `ret_wup` system call
  The `ret_wup` system call issues a wake-up request to the task specified by the parameter, and then performs return from the directly activated interrupt handler.

When a system call (`chg_pri`, `sig_sem`, etc.) that requires task scheduling is issued during the processing of a directly activated interrupt handler, the RX850 merely queues the tasks into the queue. The actual processing of task scheduling is batched and deferred until return from the directly activated interrupt handler has been completed (by issuing a `ret_int` or `ret_wup` system call).

Caution The `ret_int` and `ret_wup` system calls do 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 a directly activated interrupt handler that was initiated by an external interrupt request, notification of the termination of interrupt handler operation must be posted to the relevant external interrupt controller before any of these system calls is issued.

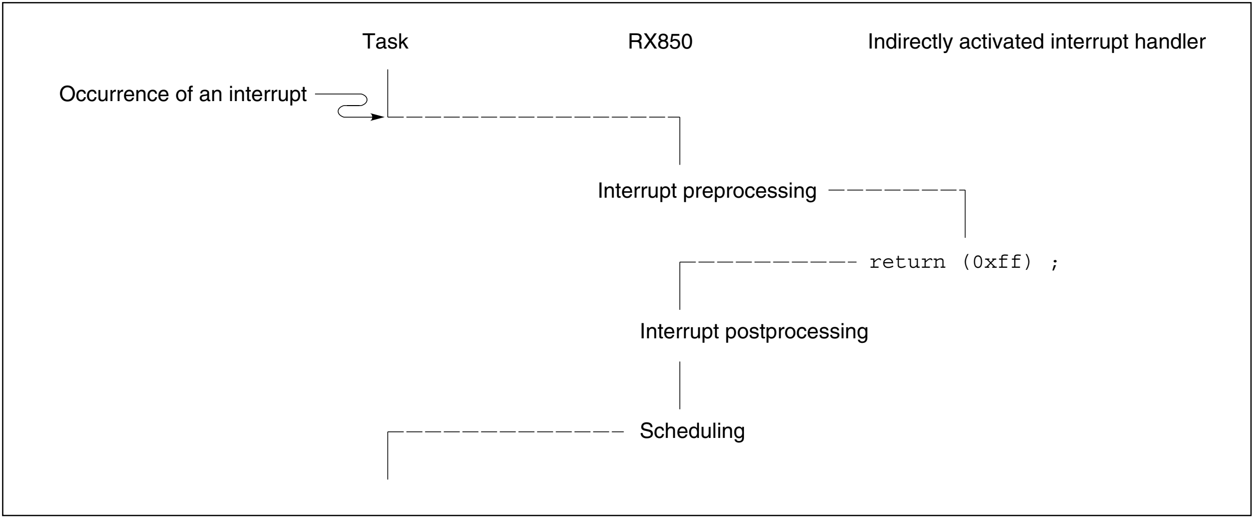
5.4 INDIRECTLY ACTIVATED INTERRUPT HANDLER

The indirectly activated interrupt handler is a routine dedicated to interrupt processing. Upon the occurrence of an interrupt, before operation of the indirectly activated interrupt handler starts, RX850 performs the required preparatory operations (such as saving the registers and stack switching).

Although an indirectly activated interrupt handler is inferior to a directly activated interrupt handler in terms of response speed, it has the advantage of the processing in the handler being simpler because the RX850 performs preprocessing of the interrupt.

Figure 5-3 shows the flow of the processing performed by the indirectly activated interrupt handler.

Figure 5-3. Flow of Processing Performed by the Indirectly Activated Interrupt Handler



5.4.1 Registering an Indirectly Activated Interrupt Handler

The indirectly activated interrupt handlers to be used by the RX850 are specified at configuration. They are registered with the RX850 only statically, based on the information specified at configuration. They cannot be registered dynamically (using system calls).

The following indirectly activated interrupt handler information is specified at configuration:

- Interrupt level
- Activation address of an indirectly activated interrupt handler

5.4.2 Internal Processing Performed by the Indirectly Activated Interrupt Handler

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

(1) Saving and Restoring the contents of a register

Based on the function call protocol for C compilers (CA850 or CCV850), the RX850 saves the work registers when control is passed to the indirectly activated interrupt handler, and restores them upon return from the interrupt handler. Therefore, the indirectly activated interrupt handler does not have to save the work registers when it starts, nor perform restoration upon the completion of its processing. Save/restoration of the registers should not, therefore, be described in the coding for the indirectly activated interrupt handler.

Caution The RX850 does not switch the gp, tp, and ep registers when control is passed to the indirectly activated interrupt handler.

(2) Stack switching

The RX850 performs stack switching when control is passed to the indirectly activated interrupt handler and also upon return from the interrupt handler. Therefore, the indirectly activated 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 indirectly activated interrupt handler.

If the interrupt handler stack is not defined during configuration, however, stack switching is not performed by the RX850. In this case, the system continues to use the stack being used upon the occurrence of the interrupt.

(3) Limitations imposed on system calls

The following lists the system calls that can be issued during the processing of an indirectly activated interrupt handler:

- Task management system calls

```
sta_tsk     chg_pri     rot_rdq     rel_wai     get_tid
ref_tsk
```

- 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     vset_flg1   vclr_flg1   vpol_flg1
vref_flg1   snd_msg     prcv_msg    ref_mbx
```

- Interrupt management system calls

```
dis_int     ena_int     chg_icr     ref_icr
```

- Memory pool management system calls

```
pget_blf    rel_blf     ref_mpf     pget_blk    rel_blk
ref_mpl
```

- Time management system calls

```
act_cyc     ref_cyc
```

- System management system calls

```
get_ver     ref_sys
```

(4) Return processing from the indirectly activated interrupt handler

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

- `return (0xff)` 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 indirectly activated interrupt handler, the RX850 merely queues the tasks into the queue. The actual processing of task scheduling is batched and deferred until return from the indirectly activated 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 indirectly activated interrupt handler that was initiated by an external interrupt request, notification of the termination of interrupt handler operation must be posted to the relevant external interrupt controller before the `return` instruction is issued.

5.5 DISABLING/RESUMING MASKABLE INTERRUPT ACCEPTANCE

The RX850 provides a function for disabling or resuming the acceptance of maskable interrupts, so that whether maskable interrupts are accepted can be specified from a user processing program.

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

(1) `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.

(2) `unl_cpu` system call

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

With the issue of this system call maskable interrupt acceptance, that was previously disabled by the issue of a `loc_cpu` system call, is enabled and dispatch processing resumes.

(3) `dis_int` system call

The `dis_int` system call disables the acceptance of maskable interrupts.

Once this system call has been issued, control is not passed to the handler until the `ena_int` system call is issued.

(4) `ena_int` system call

The `ena_int` system call resumes the acceptance of maskable interrupts.

With the issue of this system call, the acceptance of maskable interrupts, which has been disabled by the issue of the `dis_int` system call, is resumed.

Figure 5-4 shows the control flow during normal operation. Figure 5-5 shows the control flow after the issue of a `loc_cpu` system call. Figure 5-6 shows the control flow after the issue of a `dis_int` system call.

Figure 5-4. Control Flow During Normal Operation

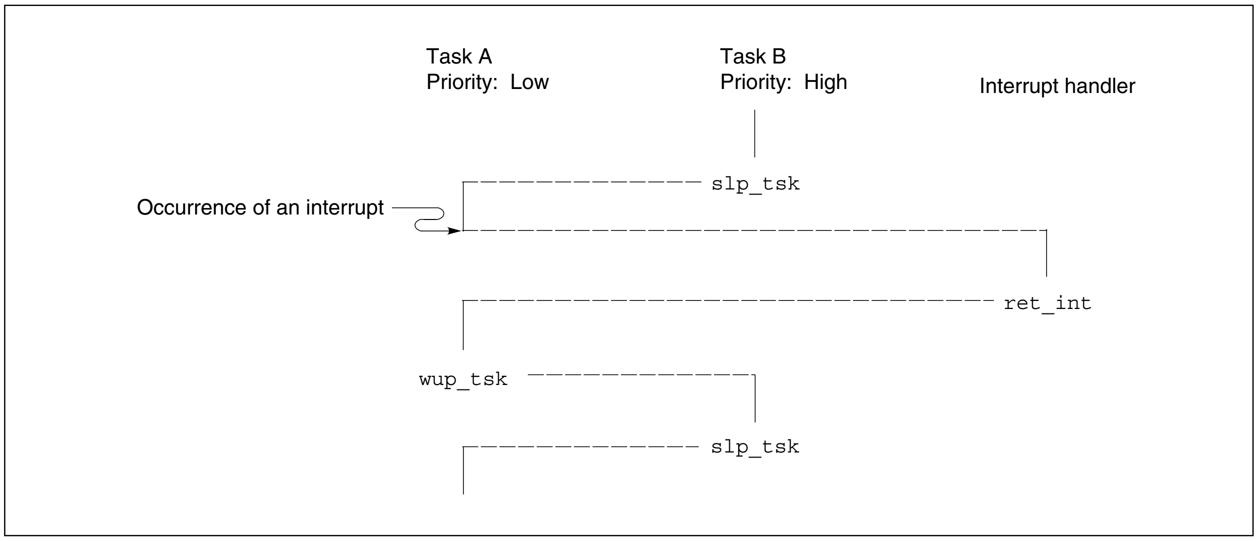


Figure 5-5. Control Flow When the `loc_cpu` System Call Is Issued

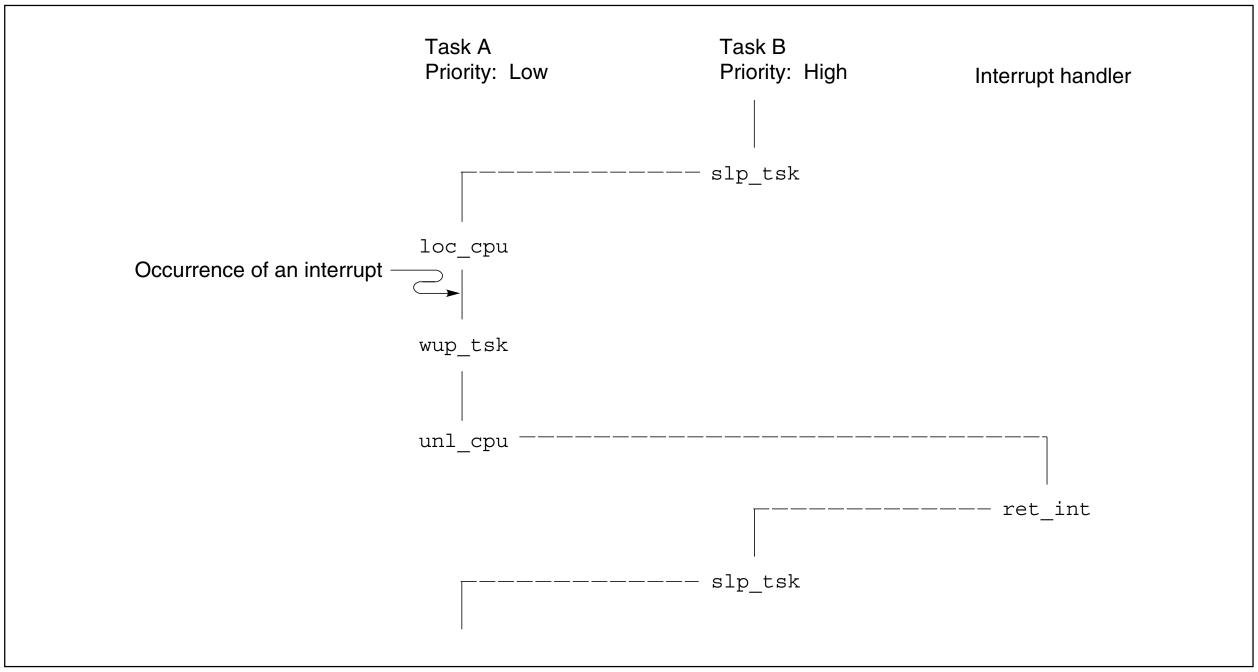
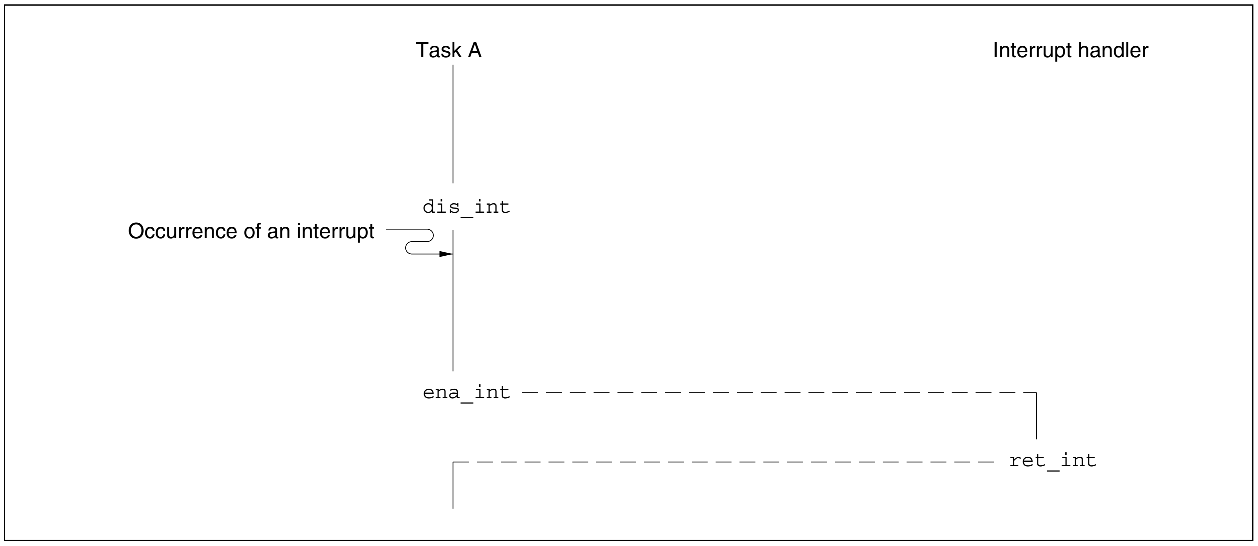


Figure 5-6. Control Flow When the `dis_int` System Call Is Issued



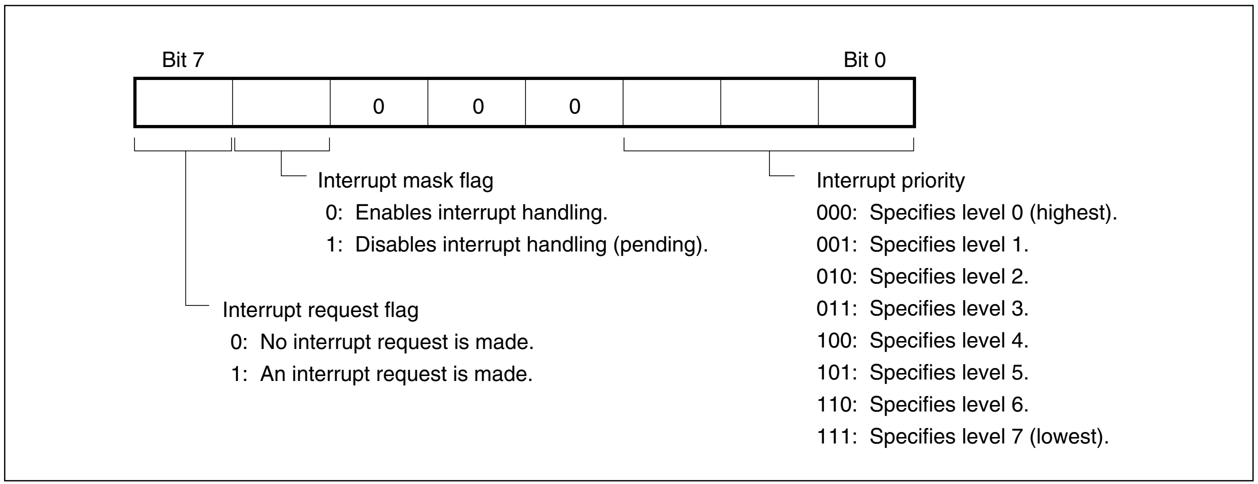
Caution System calls `ena_int` and `dis_int` enable and disable an interrupt in the task that issued these system calls. For example, suppose a maskable interrupt occurs after `dis_int` has been issued in task A. The interrupt processing remains pending. If, however, the processing moves to task B because a system call that causes dispatch is issued, and if the maskable interrupt is not disabled in task B (i.e., if system call `dis_int` is not issued in task B), the interrupt is acknowledged. This means that the interrupt processing kept pending in task A is executed immediately after the processing has been transferred to task B.
If the processing is dispatched to task A again, execution is resumed with the maskable interrupt disabled.

5.6 CHANGING/ACQUIRING THE CONTENTS OF AN INTERRUPT CONTROL REGISTER

The contents of an interrupt control register are changed using `chg_icr` and acquired using `ref_icr`.

- `chg_icr` system call
  The `chg_icr` system call changes the contents of the interrupt control register specified by the parameter.
- `ref_icr` system call
  The `ref_icr` system call acquires the contents of the interrupt control register specified by the parameter.
  Figure 5-7 shows the acquired contents of an interrupt control register.

Figure 5-7. Contents of an Interrupt Control Register



5.7 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 (the ID flag of PSW is set).

Therefore, even while processing is being executed by the RX850 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 nonmaskable interrupts, its operation cannot be assured under the RX850.

5.8 CLOCK INTERRUPTS

The RX850 performs time management, using clock interrupts that are based on the clocks supplied by the hardware (such as a real-time pulse unit) at regular intervals.

Upon the occurrence of a clock interrupt, the time management interrupt handler (clock handler) of the RX850 may be called, so that time-related processing may be performed to leave a task in the wait state for a given period and subsequently release it, initiate a cyclic handler, and so on.

For details of time management, see **Chapter 7**.

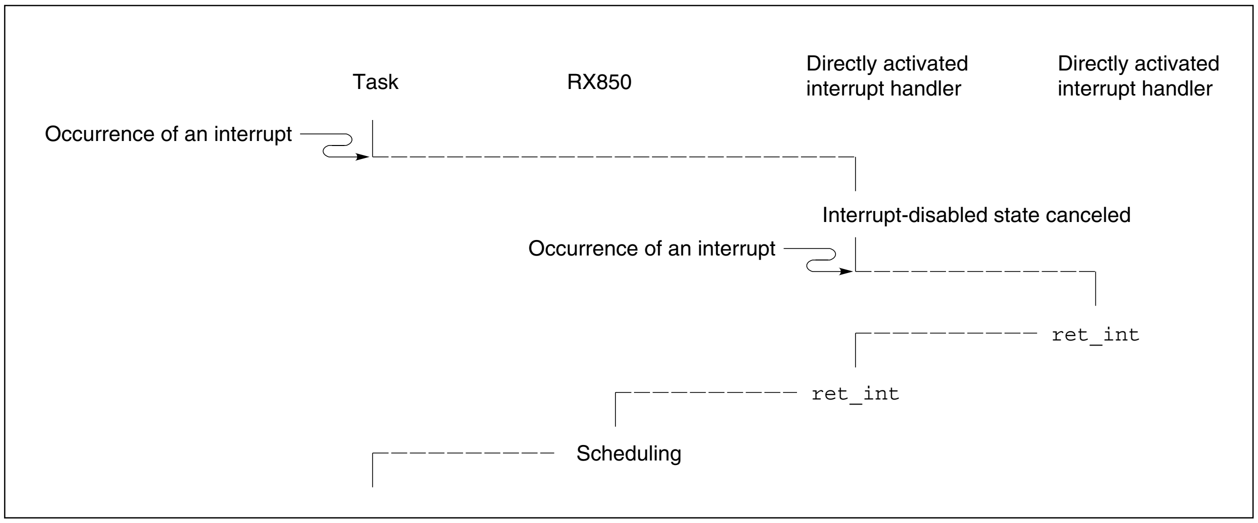
5.9 MULTIPLE INTERRUPTS

In the RX850, the occurrence of another interrupt while processing is being performed by an interrupt handler is called "**multiple interrupts**."

All interrupt handlers, however, start their operation in the interrupt-disabled state (the ID flag of PSW is set). To enable the acceptance of multiple interrupts, the canceling of the interrupt disabled state should be described in the interrupt handler.

Figure 5-8 shows the flow of the processing for handling multiple interrupts.

Figure 5-8. Flow of Processing for Handling Multiple Interrupts



Task
RX850
Directly activated interrupt handler
Directly activated interrupt handler
Occurrence of an interrupt
Interrupt-disabled state canceled
Occurrence of an interrupt
ret_int
ret_int
Scheduling

[MEMO]

CHAPTER 6   MEMORY POOL MANAGEMENT FUNCTION

This chapter describes the memory pool management function of the RX850.

6.1 OVERVIEW

The RX850 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).

The RX850 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

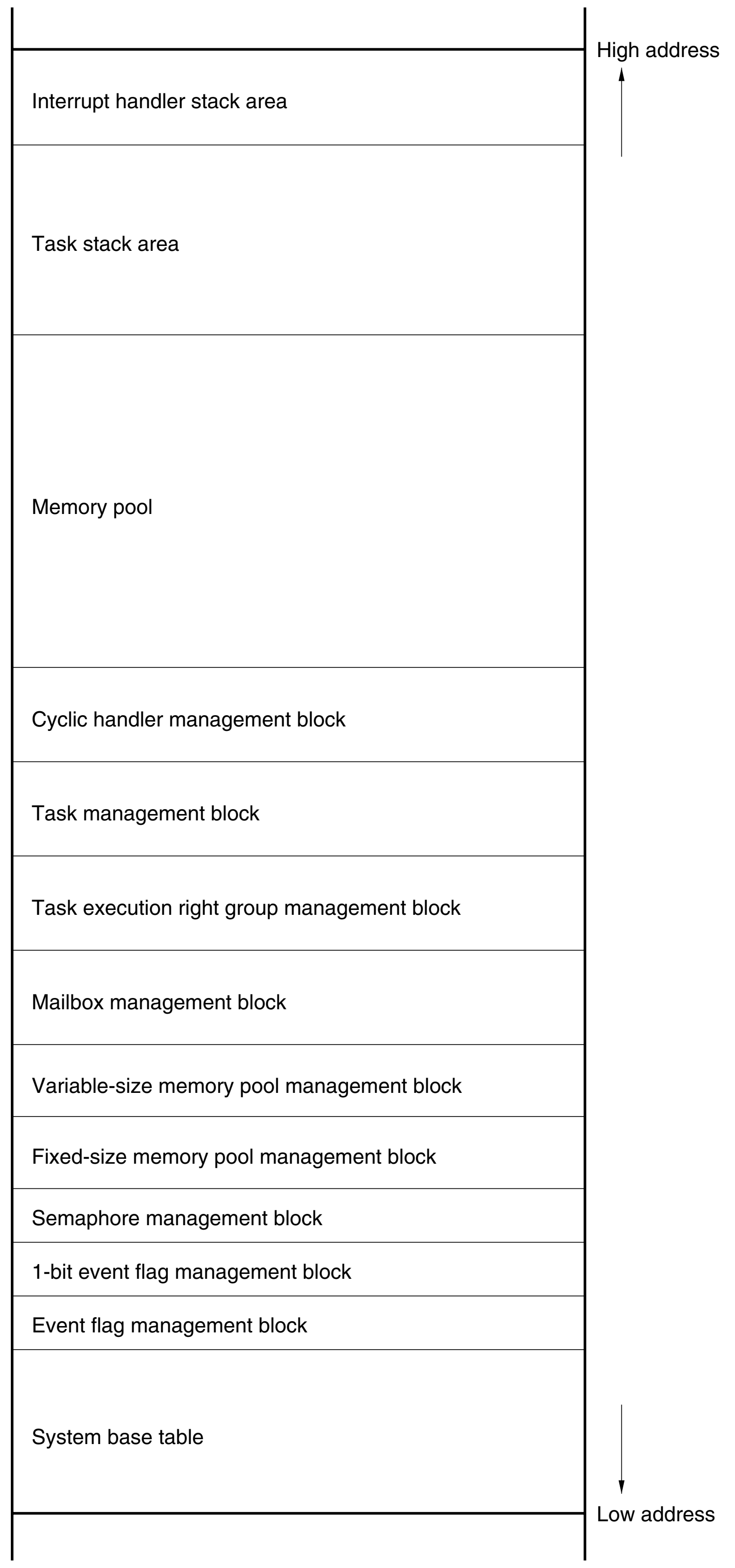
The following lists the management objects required for managing the entire system with the RX850, as well as those required for implementing the functions provided by the RX850.

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

- System base table
- Task execution right group management block
- Task management block
- Semaphore management block
- Event flag management block
- 1-bit event flag management block
- Mailbox management block
- Fixed-size memory pool management block
- Variable-size memory pool management block
- Cyclic handler management block
- Memory pool
- Task stack area
- Interrupt handler stack area

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

Figure 6-1. Typical Arrangement of Management Blocks


High address
Interrupt handler stack area
Task stack area
Memory pool
Cyclic handler management block
Task management block
Task execution right group management block
Mailbox management block
Variable-size memory pool management block
Fixed-size memory pool management block
Semaphore management block
1-bit event flag management block
Event flag management block
System base table
Low address

6.3 FIXED-SIZE MEMORY POOL

The RX850 provides a pool of fixed-size memory areas that can be used by processing programs (such as tasks and handlers).

The fixed-size memory pool consists of more than one fixed-size memory block. Operations on the fixed-size memory pool are performed in units of blocks.

Dynamic operations for the fixed-size memory pool are performed using the following fixed-size memory pool system calls.

`get_blf` :	Acquires a fixed-size memory block.
`pget_blf`:	Acquires a fixed-size memory block (by polling).
`tget_blf`:	Acquires a fixed-size memory block (with timeout setting).
`rel_blf` :	Returns a fixed-size memory block.

Cautions 1. Fixed-size memory blocks are acquired and returned in units of blocks.

2. For system calls related to the fixed-size memory pools, an ID number is used to specify the fixed-size memory pool to be manipulated (target fixed-size memory pool). Configurater CF850 consecutively assigns integers, starting with `0x1`, to fixed-size memory pools as ID numbers, based on fixed-size memory pool information stored in the system information table.

6.3.1 Fixed-Size Memory Pool Generation

The RX850 specifies the fixed-size memory pools to be used by the system at configuration. In other words, the RX850 fixed-size memory pools can be generated only statically (at system initialization) according to the information specified at configuration. They cannot be generated dynamically using system calls.

RX850 fixed-size memory pool generation consists of the acquisition and initialization of a management object (area used to manage fixed-size memory pools) and a fixed-size memory pool area.

The RX850 recognizes such areas as being fixed-size memory pools and manages them accordingly.

The following fixed-size memory pool information is specified at configuration:

- Fixed-size memory pool name
- Fixed-size memory pool size
- Type of system memory in which fixed-size memory pools are allocated (`.pool0` or `.pool1` section)

6.3.2 Acquiring a Fixed-Size Memory Block

A fixed-size memory block is acquired by issuing a `get_blf`, `pget_blf`, or `tget_blf` system call.

(1) `get_blf` system call

Upon the issue of the `get_blf` system call, the processing program (task) acquires a fixed-size memory block from the fixed-size memory pool specified by a parameter.
After the issue of this system call, if the task cannot acquire the block from the specified fixed-size memory pool (because no free block exists), the task itself is enqueued at the end of the queue of this fixed-size memory pool. Thus, the task leaves the `run` state and enters the `wait` state (the fixed-size memory block wait state).
The task shall be released from the fixed-size memory block wait state and return to the `ready` state in the following cases:

- When a `rel_blf` system call is issued.
- When a `rel_wai` system call is issued and the fixed-size memory block wait state is forcibly canceled.

Cautions 1. Tasks are queued into the queue of the specified fixed-size memory pool in the order (FIFO) in which the tasks make fixed-size memory block acquisition requests.
2. The RX850 clears only the first four bytes of any acquired fixed-size memory block. Thus, the contents of the subsequent bytes will be undefined.

(2) `pget_blf` system call

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

Caution The RX850 clears only the first four bytes of any acquired fixed-size memory block. Thus, the contents of the subsequent bytes will be undefined.

(3) `tget_blf` system call

By issuing a `tget_blf` system call, the processing program (task) acquires a fixed-size memory block from the fixed-size memory pool specified by a parameter.
After the issue of this system call, if the task cannot acquire the block from the specified fixed-size memory pool (because no free block exists), the task itself is enqueued at the end of the queue of this fixed-size memory pool. Thus, the task leaves the `run` state and enters the `wait` state (the fixed-size memory block wait state).
The task shall be released from the fixed-size memory block wait state and return to the `ready` state in the following cases:

- Once the wait time specified by a parameter has elapsed.
- When a `rel_blf` system call is issued.
- When a `rel_wai` system call is issued and the fixed-size memory block wait state is forcibly canceled.

Cautions 1. Tasks are queued into the queue of the specified fixed-size memory pool in the order (FIFO) in which the tasks make fixed-size memory block acquisition requests.
2. The RX850 clears only the first four bytes of any acquired fixed-size memory block. Thus, the contents of the subsequent bytes will be undefined.

6.3.3 Returning a Fixed-Size Memory Block

A fixed-size memory block is returned upon the issue of a `rel_blf` system call.

(1) `rel_blf`

Upon the issue of a `rel_blf` system call, a processing program (task) returns a fixed-size memory block to the fixed-size memory pool specified by a parameter.

When this system call is issued, if any tasks are queued into the queue of the specified fixed-size memory pool, the fixed-size memory block is not returned, but is passed to the first task in the queue.

Thus, the first task is removed from the queue, leaves the `wait` state (the fixed-size 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 fixed-size memory block are not cleared automatically by the RX850. Thus, the contents of a fixed-size memory block may be undefined when that memory block is returned.

2. A memory block shall be returned to the same fixed-size memory pool as that specified by the `get_blf`, `pget_blf`, or `tget_blf` system call.

6.3.4 Acquiring Fixed-Size Memory Pool Information

Fixed-size memory pool information is acquired by issuing a `ref_mpf` system call.

(1) `ref_mpf` system call

Upon the issue of a `ref_mpf` system call, the processing program (task) acquires the fixed-size memory pool information (extended information, queued tasks, etc.) for the fixed-size memory pool specified by a parameter.

The fixed-size memory pool information consists of the following:

- Extended information
- Whether tasks are queued

`FALSE(0)`	:	No task is queued.
Value	:	ID number of the first task in the queue

- Number of free blocks

6.4 VARIABLE-SIZE MEMORY POOL

The RX850 provides a pool of variable-size memory areas that can be used by processing programs (such as tasks and handlers).

The variable-size memory pool consists of more than one variable-size memory block. Operations on the variable-size memory pool are performed in units of blocks.

Dynamic operations for the variable-size memory pool are performed using the following variable-size memory pool system calls.

`get_blk` :	Acquires a variable-size memory block.
`pget_blk`:	Acquires a variable-size memory block (by polling).
`tget_blk`:	Acquires a variable-size memory block (with timeout setting).
`rel_blk` :	Returns a variable-size memory block.

Cautions 1. Variable-size memory blocks are acquired and returned in units of blocks.

2. For system calls related to the variable-size memory pools, an ID number is used to specify the variable-size memory pool to be manipulated (target variable-size memory pool). Configurater CF850 consecutively assigns integers, starting with `0x1`, to variable-size memory pools as ID numbers, based on variable-size memory pool information stored in the system information table.

6.4.1 Variable-Size Memory Pool Generation

The RX850 specifies the variable-size memory pools to be used by the system at configuration. In other words, the RX850 variable-size memory pools can be generated only statically (at system initialization) according to the information specified at configuration. They cannot be generated dynamically using system calls.

RX850 variable-size memory pool generation consists of the acquisition and initialization of a management object (area used to manage variable-size memory pools) and a variable-size memory pool area.

The RX850 recognizes such areas as being variable-size memory pools and manages them accordingly.

The following variable-size memory pool information is specified at configuration:

- Variable-size memory pool name
- Variable-size memory pool size
- Type of system memory in which variable-size memory pools are allocated (`.pool0` or `.pool1` section)

6.4.2 Acquiring a Variable-Size Memory Block

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

(1) `get_blk` system call

Upon the issue of the `get_blk` system call, the processing program (task) acquires a variable-size memory block from the variable-size memory pool specified by a parameter.

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

The task shall be released from the variable-size memory block wait state and return to the `ready` state in the following cases:

- When a `rel_blk` system call is issued.
- When a `rel_wai` system call is issued and the variable-size memory block wait state is forcibly canceled.

Cautions 1. Tasks are queued into the queue of the specified variable-size memory pool in the order (FIFO) in which the tasks make variable-size memory block acquisition requests.

2. The RX850 clears only the first four bytes of any acquired variable-size memory block. Thus, the contents of the subsequent bytes will be undefined.

(2) `pget_blk` system call

Upon the issue of the `pget_blk` system call, the processing program (task) acquires a variable-size memory block from the variable-size memory pool specified by a parameter.

For this system call, if the task cannot acquire the block from the variable-size memory pool specified by this system call parameter (because no free block exists), `E_TMOUT` is returned as the return value.

Caution The RX850 clears only the first four bytes of any acquired variable-size memory block. Thus, the contents of the subsequent bytes will be undefined.

(3) `tget_blk` system call

Upon the issue of the `tget_blk` system call, the processing program (task) acquires a variable-size memory block from the variable-size memory pool specified by a parameter.

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

The task shall be released from the variable-size memory block wait state and return to the `ready` state in the following cases:

- Once the wait time specified by a parameter has elapsed.
- When a `rel_blk` system call is issued.
- When a `rel_wai` system call is issued and the variable-size memory block wait state is forcibly canceled.

Cautions 1. Tasks are queued into the queue of the specified variable-size memory pool in the order (FIFO) in which the tasks make variable-size memory block acquisition requests.

2. The RX850 clears only the first four bytes of any acquired variable-size memory block. Thus, the contents of the subsequent bytes will be undefined.

6.4.3 Returning a Variable-Size Memory Block

A variable-size memory block is returned upon the issue of a `rel_blk` system call.

(1) `rel_blk`

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

When this system call is issued, if any tasks are queued into the queue of the specified variable-size memory pool, the variable-size memory block is not returned, but is passed to the first task in the queue.

Thus, the first task is removed from the queue, leaves the `wait` state (the variable-size 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 variable-size memory block are not cleared automatically by the RX850. Thus, the contents of a variable-size memory block may be undefined when that memory block is returned.

2. A memory block shall be returned to the same variable-size memory pool as that specified by the `get_blk`, `pget_blk`, or `tget_blk` system call.

6.4.4 Acquiring Variable-Size Memory Pool Information

Variable-size memory pool information is acquired by issuing a `ref_mpl` system call.

(1) `ref_mpl` system call

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

The variable-size memory pool information consists of the following:

- Extended information
- Whether tasks are queued

`FALSE(0)`	:	No task is queued.
Value	:	ID number of the first task in the queue

- Number of free blocks

CHAPTER 7 TIME MANAGEMENT FUNCTION

This chapter describes the time management function of the RX850.

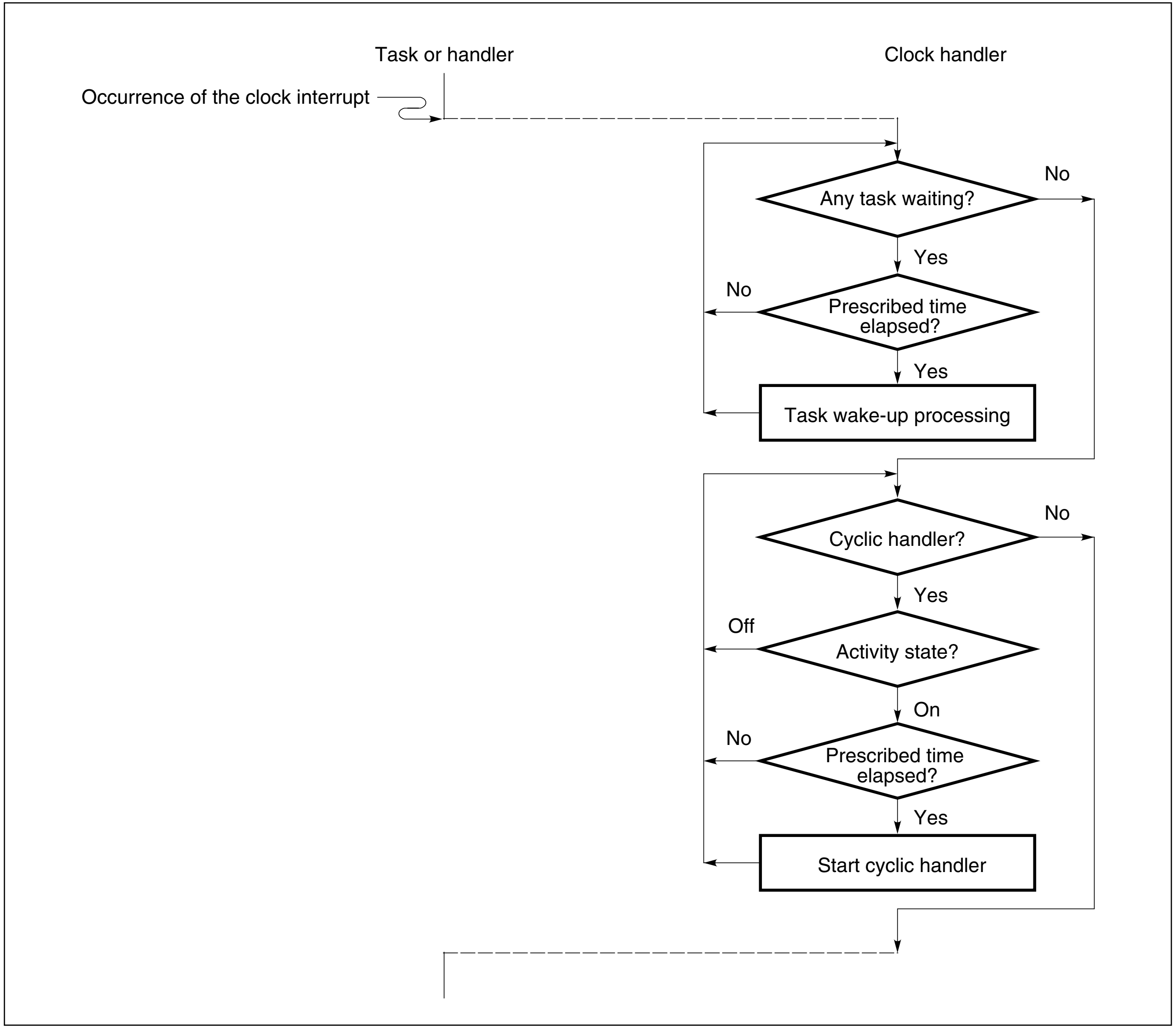
7.1 OVERVIEW

The RX850 uses a clock interrupt, generated by hardware (real-time pulse unit) at regular intervals, for time management.

Each time a clock interrupt occurs, the time management interrupt handler of the RX850 (clock handler) starts and controls processing such as managing the timeout wait state of tasks and starting the cyclic handler.

Figure 7-1 shows the processing performed by the clock handler.

Figure 7-1. Clock Handler Processing



7.2 TIMER OPERATION

Real-time processing requires a clock synchronization function (based on timer operation) to suspend the processing of a task for a given period, perform periodic handler processing, and so on. RX850 provides functions, based on timer operation, such as delayed task wake-up, timeout, and the initiation of a cyclic handler.

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.

(1) `dly_tsk` system call

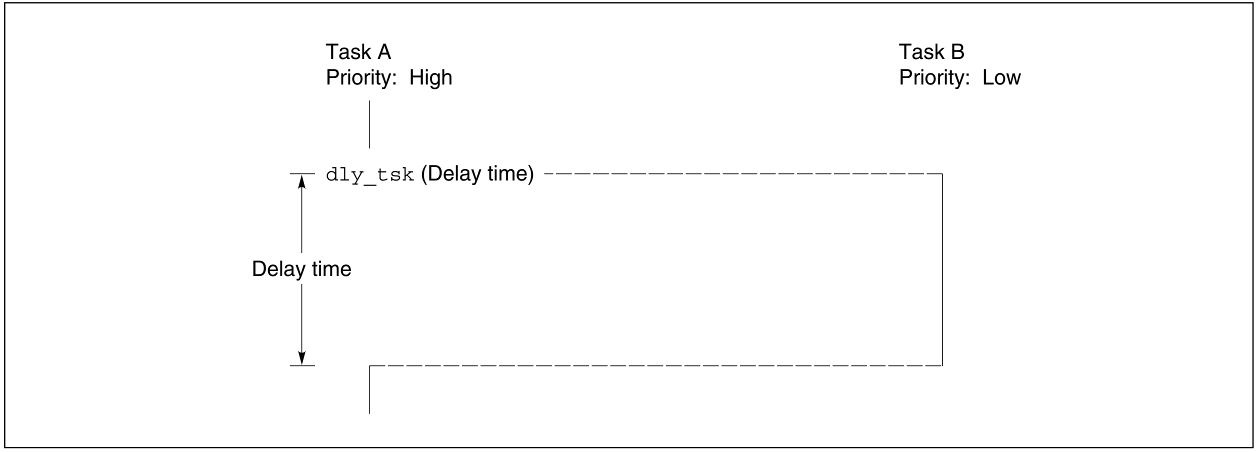
Upon the issue of a `dly_tsk` system call, the state of the task from which this system call was issued changes from `run` to `wait` (the timeout wait state).

The task shall be released from the timeout wait state and return to the `ready` state in the following cases:

- 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 timeout wait state.

Figure 7-2 shows the flow of the processing after the issue of the `dly_tsk` system call.

Figure 7-2. 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`, `vtwai_flg1`, `trcv_msg`, `tget_blf`, or `tget_blk` system call.

(1) `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 (the wake-up request counter is not decremented) and the task enters the `wait` state (the wake-up wait state) from the `run` state.

The task shall be released from the wake-up wait state and return to the `ready` state in the following cases:

- **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 wake-up wait state is forcibly canceled.

(2) `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 at the end of the queue of this semaphore. Thus, the task leaves the `run` state and enters the `wait` state (the resource wait state).

The task shall be released from this resource wait state and return to the `ready` state in the following cases:

- **When the given wait time specified by a parameter has elapsed.**
- When a `sig_sem` system call is issued.
- When a `rel_wai` system call is issued and the resource wait state is forcibly canceled.

(3) `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 into the queue of this event flag. Thus, the task leaves the `run` state and enters the `wait` state (the event flag wait state).

The task shall be released from this event flag wait state and return to the `ready` state in the following cases:

- **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 `rel_wai` system call is issued and the event flag wait state is forcibly canceled.

(4) `vtwai_flg1` system call

The `vtwai_flg1` system call checks whether `1` is set in the 1-bit event flag specified by a parameter.

If `1` is not set in the 1-bit 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 task shall be released from this event flag wait state and return to the `ready` state in the following cases:

- **When the given wait time specified by a parameter has elapsed.**
- When a `vset_flg1` system call is issued.
- When a `rel_wai` system call is issued and the event flag wait state is forcibly canceled.

(5) `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), 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 task shall be released from the message wait state and return to the `ready` state in the following cases:

- **When the given time specified by a parameter has elapsed.**
- When a `snd_msg` system call is issued.
- When a `rel_wai` system call is issued and the message wait state is forcibly canceled.

(6) `tget_blf` system **call**

Upon the issue of a `tget_blf` system call, the task acquires a fixed-size memory block from the fixed-size memory pool specified by a parameter.

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

The task shall be released from this fixed-size memory block wait state and return to the `ready` state in the following cases:

- **When the given wait time specified by a parameter has elapsed.**
- When a `rel_blf` system call is issued.
- When a `rel_wai` system call is issued and the fixed-size memory block wait state is forcibly canceled.

(7) `tget_blk` system call

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

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

The task shall be released from this variable-size memory block wait state and return to the `ready` state in the following cases:

- **When the given wait time specified by a parameter has elapsed.**
- When a `rel_blk` system call is issued.
- When a `rel_wai` system call is issued and the variable-size memory block wait state is forcibly canceled.

7.5 CYCLIC HANDLER

A cyclic handler is a routine dedicated to cyclic processing, initiated at regular intervals. The cyclic handler is handled independently of all other tasks. Even if a task of the highest priority in the system is currently being executed, it is interrupted by this handler.

The cyclic handler is a processing program with the minimum overhead of the processing programs that is described by the user and which is executed cyclically, up to the start of execution.

7.5.1 Registering a Cyclic Handler

The cyclic handler to be used by the RX850 is specified at configuration. It can be registered only statically (at system initialization) based on the information specified at configuration. It cannot be registered dynamically (using system calls).

In the RX850, registration of the cyclic handler consists of the acquisition and initialization of a management object (area used to manage the cyclic handler).

The following cyclic handler information is specified at configuration:

- Name of cyclic handler
- Activation address of cyclic handler
- Initial activity state of cyclic handler
- Activation interval of cyclic handler

Caution For system calls related to the cyclic handler, an ID number is used to specify which cyclic handler is to be manipulated (target cyclic handler). Configurater CF850 consecutively assigns integers, starting with `0x1`, to cyclic handlers, as ID numbers based on cyclic handler information stored in the system information table.

7.5.2 Activity State of the Cyclic Handler

The activity state of a cyclic handler is one of the criteria which the RX850 uses to determine whether to start the cyclic handler.

The activity state of a cyclic handler is specified at system generation according to the information specified at configuration. The RX850 provides a function that enables a user processing program to dynamically change the activity state of a cyclic handler.

- `act_cyc` system call

  This system call changes the activity state of a cyclic handler specified by the parameter.

`TCY_OFF`	:	Switches the activity state of the cyclic handler to OFF.
`TCY_ON`	:	Switches the activity state of the cyclic handler to ON.
`TCY_INI`	:	Enqueues the cyclic handler into the timer queue, then initializes a cycle counter.
`TCY_ULNK`	:	Switches the activity state of the cyclic handler to OFF, then detaches the handler from the timer queue.

Under the RX850, while the cyclic handler is queued into the timer queue, the cycle counter continues to count even if the activity state of that handler is OFF. In some cases, when the `act_cyc` system call is issued to switch the activity state of the cyclic handler from OFF to ON, the first activation request could be issued sooner than the activation interval specified at configuration. To prevent this, the user must specify `TCY_INI` to initialize the cycle counter as well as `TCY_ON` to restart the cyclic handler when issuing the `act_cyc` system call. Then, the first activation request will be issued in sync with the activation interval specified at configuration.

Figures 7-3 and 7-4 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 cyclic handler from OFF to ON.

In these figures, ΔT is assumed to be the activation time interval, specified for the cyclic handler at configuration. In Figure 7-3, the relationship between Δt and ΔT is Δt ≤ ΔT.

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

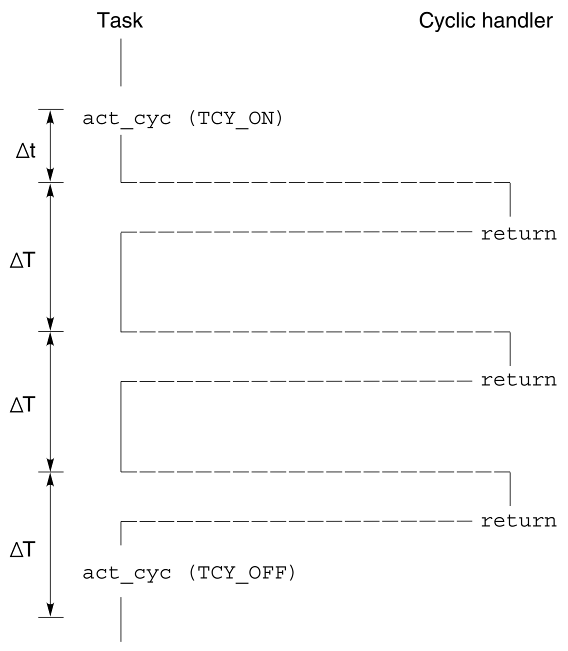
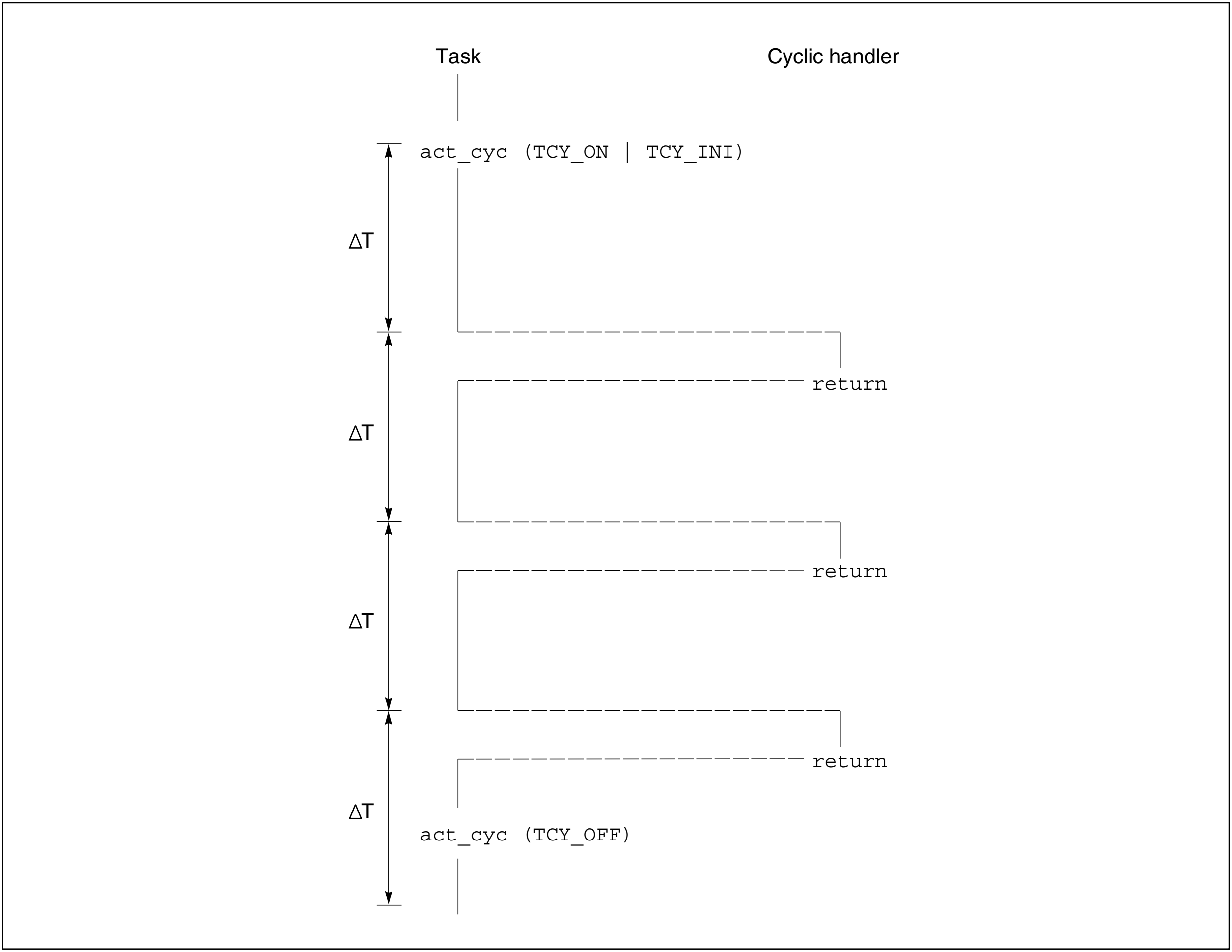


Figure 7-4. Flow of Processing After Issue of `act_cyc (TCY_ON|TCY_INI)`



7.5.3 Internal Processing Performed by the Cyclic Handler

After the occurrence of a clock interrupt, the RX850 performs preprocessing for interruption before control is passed to the cyclic handler. When control is returned from the cyclic handler, the RX850 performs interrupt post processing.

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

(1) Saving and restoring the contents of a register

Based on the function call protocol for C compilers (CA850 or CCV850), the RX850 saves the work registers when control is passed to the cyclic handler, and restores them upon the return of control from the handler. Therefore, the cyclic 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 cyclic handler.

Caution When passing control to the cyclic handler, the RX850 does not switch the `gp`, `tp`, or `ep` register.

(2) Stack switching

The RX850 performs stack switching when control is passed to the cyclic handler and upon a return from the handler. Therefore, the cyclic 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 be coded in the description of the cyclic handler.

If the interrupt handler stack is not defined during configuration, however, stack switching is not performed by the RX850. In this case, the system continues to use that stack being used upon the occurrence of an interrupt.

(3) Limitations imposed on system calls

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

- Task management system calls

`sta_tsk`	`chg_pri`	`rot_rdq`	`rel_wai`	`get_tid`
`ref_tsk`				

- 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`	`vset_flg1`	`vclr_flg1`	`vpol_flg1`
`vref_flg1`	`snd_msg`	`prcv_msg`	`ref_mbx`	

- Interrupt management system calls

`dis_int`	`ena_int`	`chg_icr`	`ref_icr`

- Memory pool management system calls

`pget_blf`	`rel_blf`	`ref_mpf`	`pget_blk`	`rel_blk`
`ref_mpl`				

- Time management system calls

`act_cyc`	`ref_cyc`

- System management system calls

`get_ver`	`ref_sys`

(4) Return processing from the cyclic handler

Return processing from the cyclic handler is performed by issuing an `return` system call upon the completion of the processing performed by the cyclic handler.

- `return` system call
  The `return` system call performs return from the cyclic handler.

When a system call (`chg_pri`, `sig_sem`, etc.) that requires task scheduling is issued during the processing of a cyclic handler, the RX850 merely queues that task into the queue. The actual task scheduling is batched and deferred until return from the cyclic handler has been completed (by issuing an `return` system call).

7.5.4 Acquiring Cyclic Handler Information

Information related to a cyclic handler is acquired by issuing a `ref_cyc` system call.

(1) `ref_cyc` system call

By issuing a `ref_cyc` system call, the task acquires information (including extended information, remaining time, etc.) related to the cyclic handler specified by a parameter.
The cyclic handler information consists of the following:

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

[MEMO]

CHAPTER 8 SCHEDULER

This chapter explains the task scheduling performed by the RX850.

8.1 OVERVIEW

By monitoring the dynamically changing task states, the RX850 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 RX850 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.

In other words, the scheduler is activated and performs task scheduling according to the issue of a system call that may cause a task state shift, the issue of a return instruction by a handler, or the occurrence of a clock interrupt.

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

- Task management system calls
```
sta_tsk     ext_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
```
sig_sem     wai_sem     twai_sem    set_flg     wai_flg
twai_flg    vset_flg1   vwai_flg1   vtwai_flg1  snd_msg
rcv_msg     trcv_msg
```

- Interrupt management system calls
```
ret_int     ret_wup     unl_cpu
```

- Memory pool management system calls
```
get_blf     tget_blf    rel_blf     get_blk     tget_blk
rel_blk
```

- Time management system calls
```
dly_tsk
```

8.3 SCHEDULING METHOD

The RX850 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 `run` or `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 `run` or `ready` state), selects the task having the highest priority, and assigns a processing time to the selected task.

Caution In the RX850, a task to which a smaller value is assigned as the priority level has a higher priority.

8.3.2 FCFS (First-Come, First-Served) Method

The RX850 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

An idle handler is a routine which is activated by the scheduler when all the tasks have left the `run` or `ready` state, that is, when there are no tasks to be scheduled by the RX850 in the system. This routine is handled independently of the tasks.

The idle handler is used to exploit the power save function of the V850 Family. For the processing of the idle handler, sample source files are provided.

8.4.1 Internal Processing Performed by the Idle Handler

The idle handler sets the power save function provided by the V850 Family. The following describes this power save function.

(1) HALT mode

In this mode, the clock generators (oscillator and PLL synthesizer) continues to operate, while the processor operating clock is no longer supplied. However, clocks are still supplied to other internal peripheral functions so that these functions continue to operate.

By combining HALT mode with normal operation mode to provide intermittent operation, the total power consumption in the system can be reduced.

To switch the system to HALT mode, the dedicated instruction (HALT instruction) must be issued.

(2) IDLE mode

In this mode, the clock generators (oscillator and PLL synthesizer) continue to operate, while the internal system clock is no longer supplied so that the entire system is stopped.

When IDLE mode is canceled, the clock settling time (such as the oscillation settling time for the oscillator) is not required. This allows the system to enter normal operation mode quickly.

IDLE mode stands midway between HALT mode and STOP mode in terms of power consumption and clock settling time. This mode can be used for those applications that require less power consumption and operation without the clock settling time.

To switch the system to IDLE mode, the control register (power save control register PSC) must be set.

(3) STOP mode

In this mode, the clock generators (oscillator and PLL synthesizer) are stopped so that the entire system is stopped.

While in STOP mode, the system consumes the least power, in that only the leakage current flows.

To switch the system to STOP mode, the control register (power save control register PSC) must be set.

8.5 IMPLEMENTING A ROUND-ROBIN METHOD

The RX850 uses the priority and FCFS scheduling methods. In scheduling based on these 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.

The RX850 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.

An example of exclusive control among tasks by using semaphores in the round-robin method is given below.

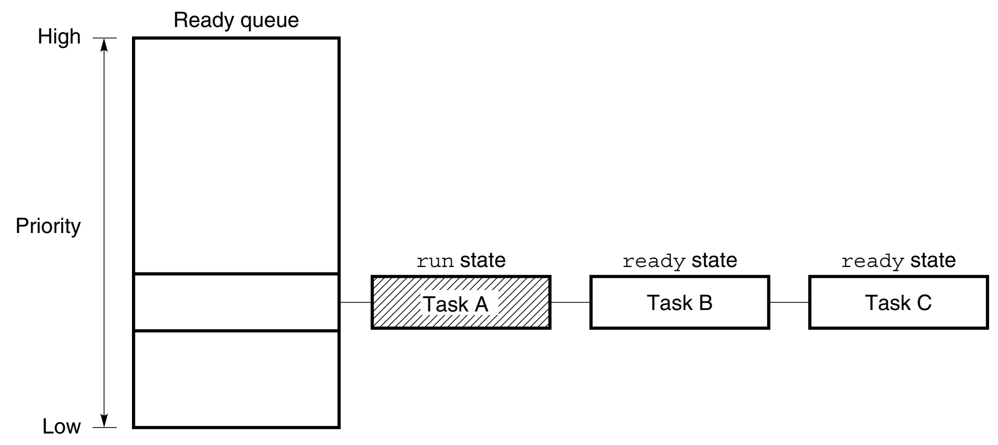
(Prerequisites)

- Task priority
  Task A = task B = task C
- State of tasks
  Task A: `run` state
  Task B: `ready` state
  Task C: `ready` state
- Cyclic handler attributes

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

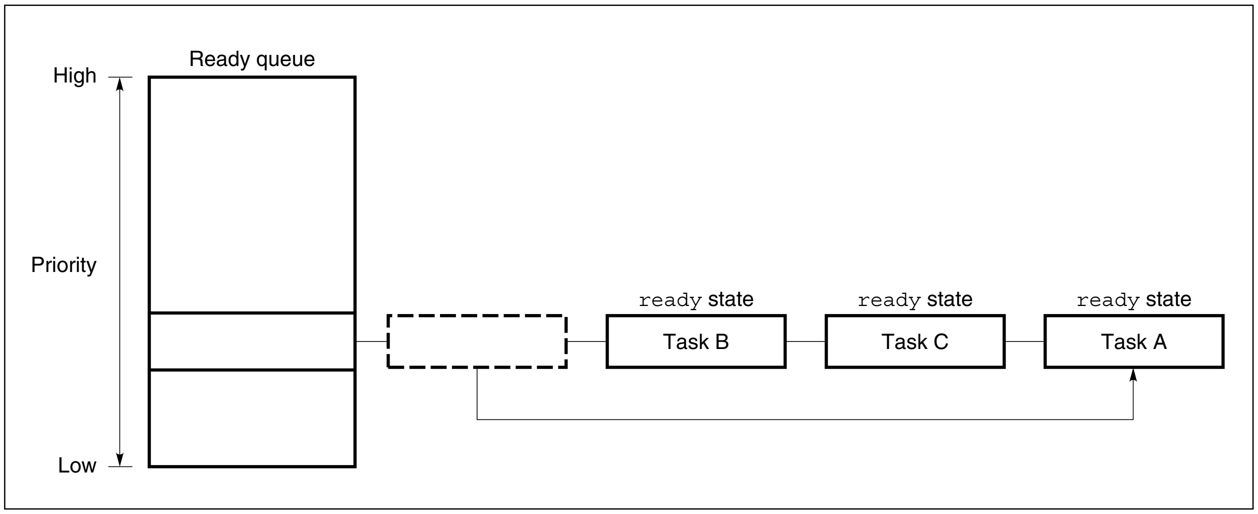
(1) Task A is currently running.
The other tasks (B and C) have the same priority as task A, but they cannot be executed unless task A enters another state or relinquishes control of the processor.
The ready queue becomes as shown in Figure 8-1.

Figure 8-1. Ready Queue State



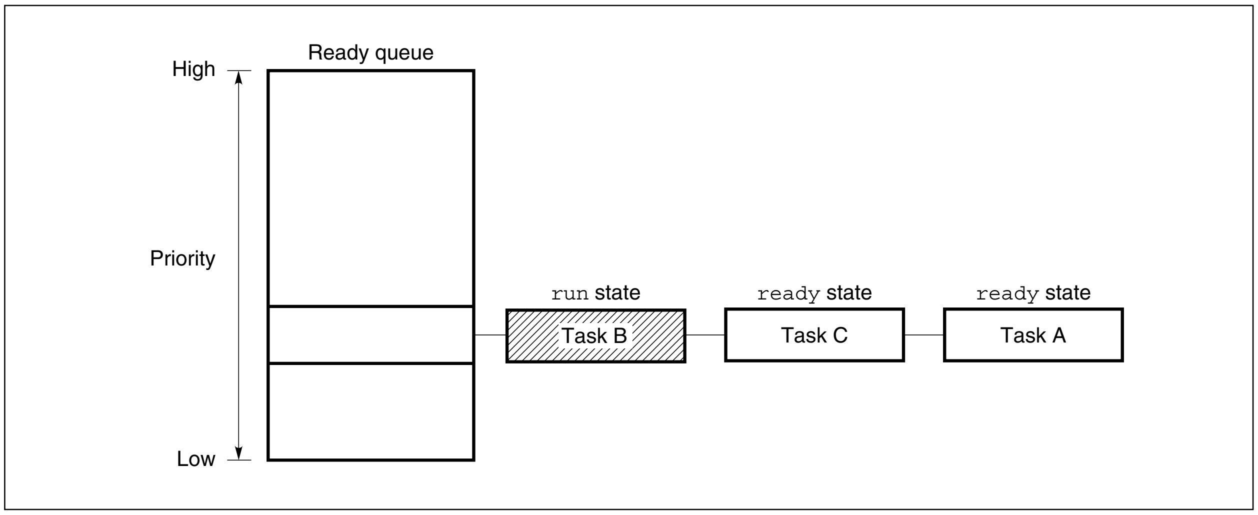
(2) With the elapse of time, the cyclic handler starts and issues the `rot_rdq` system call.
Task A is added to the end of the ready queue having the same priority, and changes from the `run` state to the `ready` state.
The ready queue changes to the state shown in Figure 8-2.

Figure 8-2. Ready Queue State



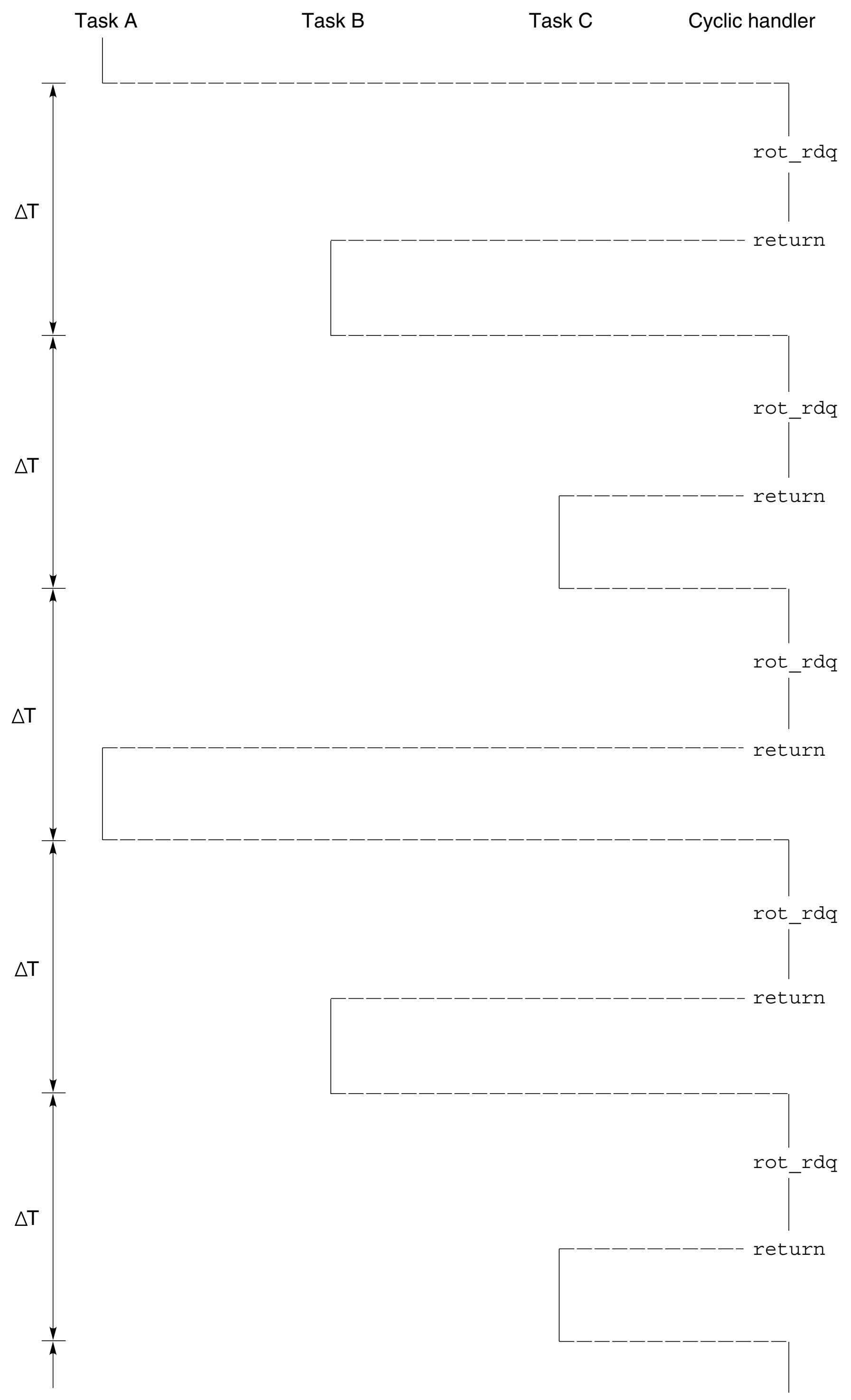
(3) As task A enters the `ready` state, task B changes from the `ready` state to the `run` state.
The ready queue changes to the state shown in Figure 8-3.

Figure 8-3. Ready Queue State



(4) By issuing the `rot_rdq` system call from the cyclic 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 8-4 shows the processing flow when the round-robin method is used.

Figure 8-4. Processing Flow When the Round-Robin Method Is Used



8.6 SCHEDULING LOCK FUNCTION

The RX850 supports that enable a user processing program to drive the scheduler and disable or resume dispatching (task scheduling).

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

(1) `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.

(2) `ena_dsp` system call

Resumes dispatching (task scheduling).

When this system call is issued, dispatching which was disabled by the issue of the `dis_dsp` system call is resumed.

(3) `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.

(4) `unl_cpu` system call

Enables the acceptance of maskable interrupts, then restarts dispatching (task scheduling).

When this system call is issued, the acceptance of maskable interrupts which was disabled by the issue of the `loc_cpu` system call is re-enabled, allowing dispatching to be resumed.

Figure 8-5 shows the normal control flow. Figure 8-6 shows the control flow when the `dis_dsp` system call is issued. Figure 8-7 shows the control flow when the `loc_cpu` system call is issued.

Figure 8-5. Control Flow During Normal Operation

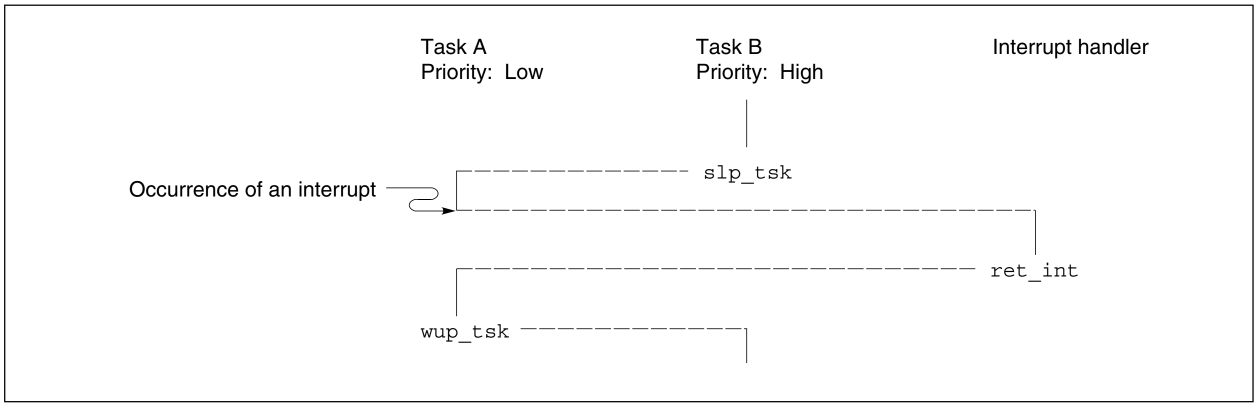


Figure 8-6. Control Flow When the `dis_dsp` System Call Is Issued

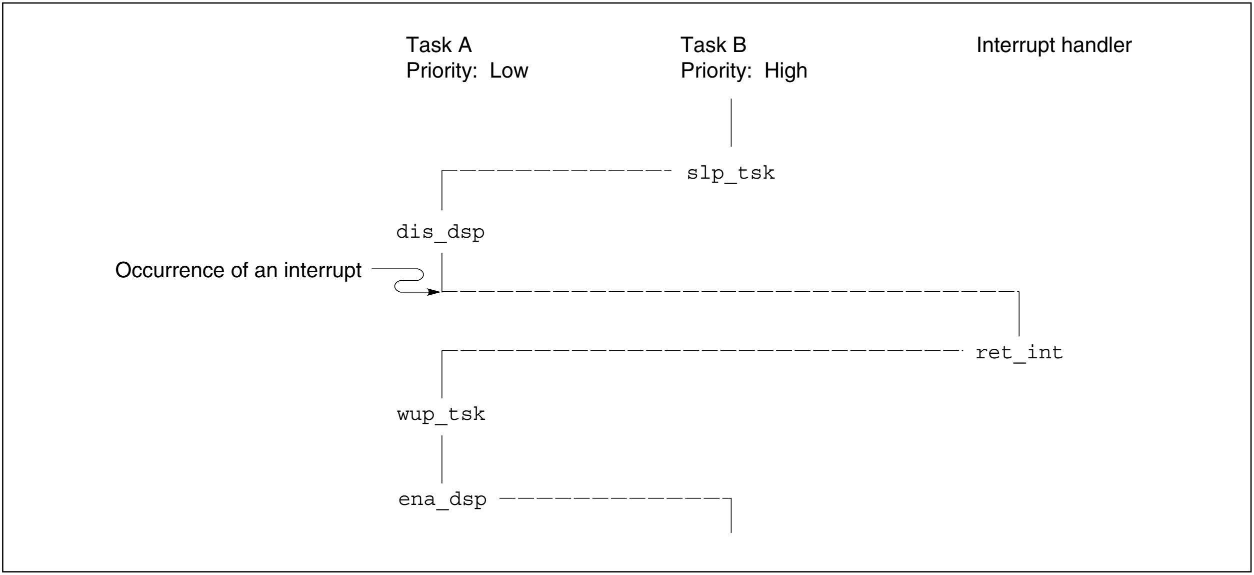
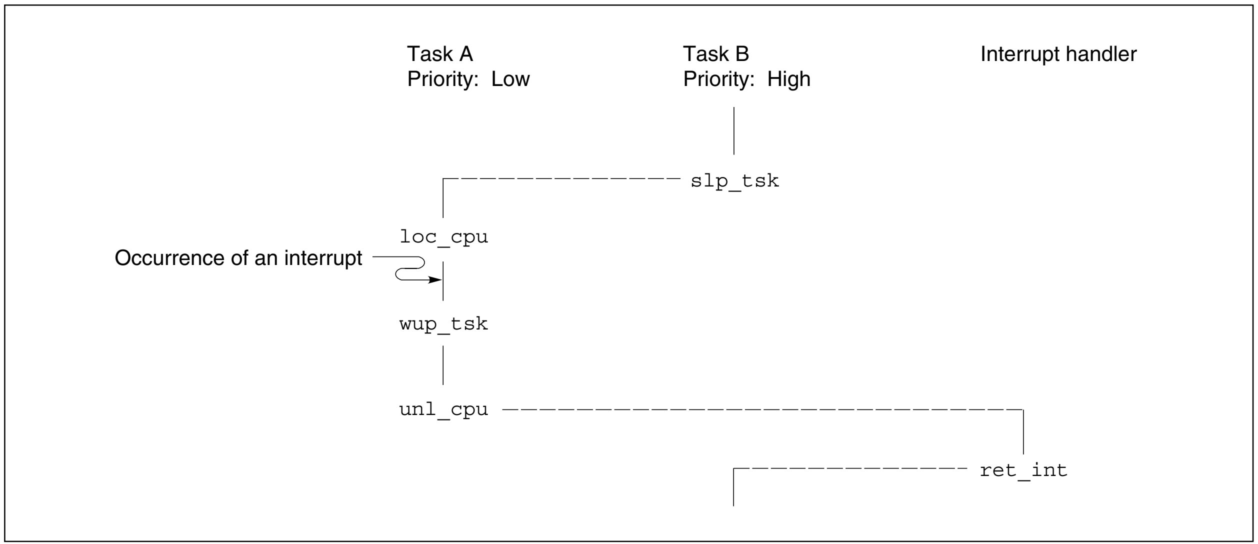


Figure 8-7. Control Flow When the `loc_cpu` System Call Is Issued



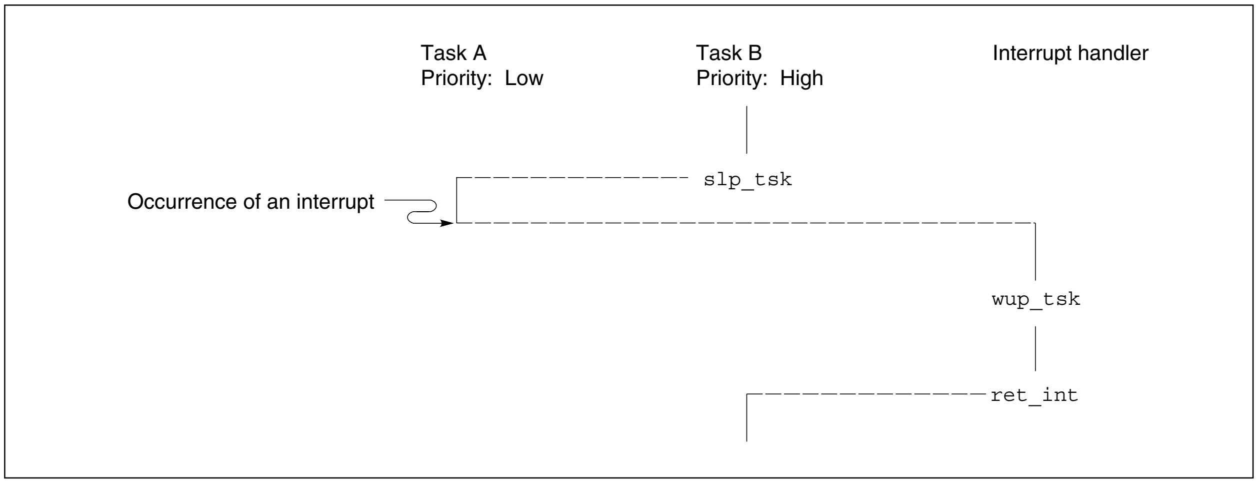
8.7 SCHEDULING WHILE THE HANDLER IS OPERATING

To quickly terminate handlers (interrupt handlers and cyclic handlers), the RX850 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, the RX850 merely executes processing such as queue operation until the completion of return processing from the handler (such as `ret_int` or `return`). Actual scheduling is delayed and executed at one time upon the completion of return processing.

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

Figure 8-8. Control Flow When the `wup_tsk` System Call Is Issued



[MEMO]

CHAPTER 9 SYSTEM INITIALIZATION

This chapter explains the system initialization to be performed by the RX850.
For details of the system initialization, refer to the **RX850 User's Manual, Installation**.

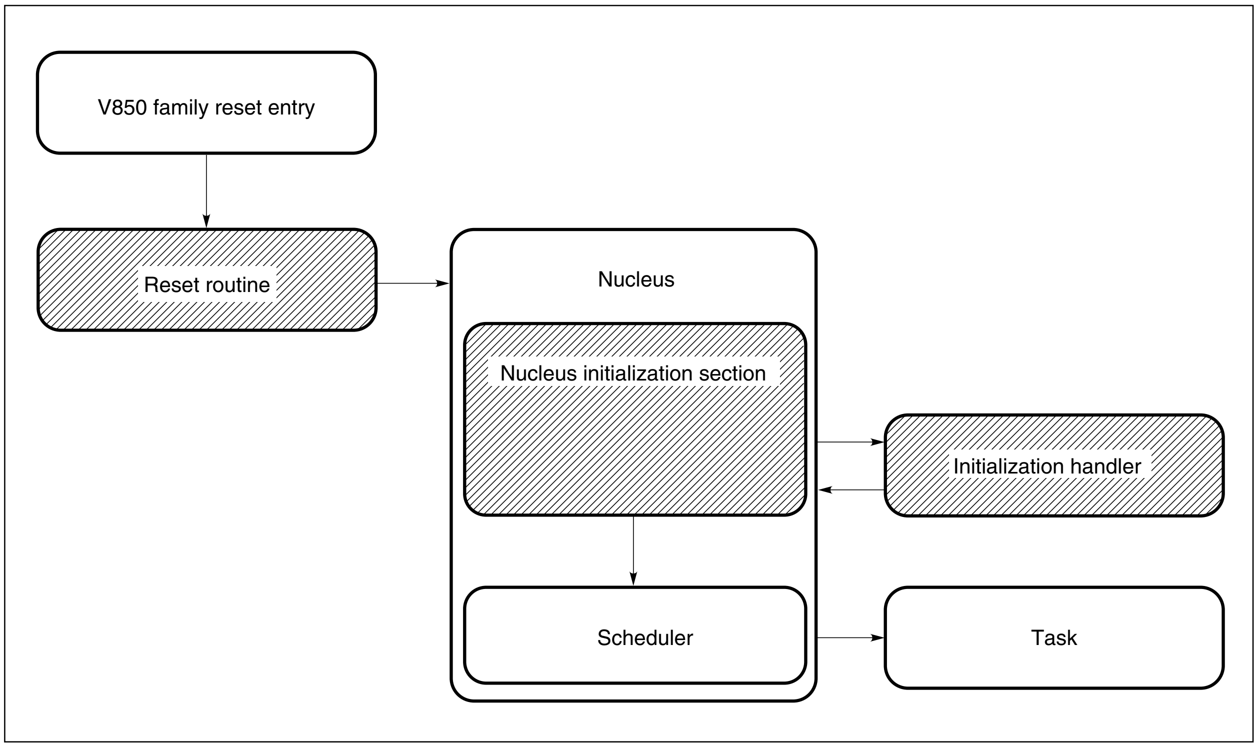
9.1 OVERVIEW

System initialization consists of initializing the hardware required by the RX850 (the reset routine), as well as initializing the software (the nucleus initialization section and initialization handler).

For this reason, in the RX850, the processing performed immediately after the system has been started is system initialization.

Figure 9-1 shows the flow of system initialization.

Figure 9-1. Flow of System Initialization



Caution System initialization is performed while the acceptance of maskable interrupts is disabled. If the acceptance of maskable interrupts is enabled during system initialization, its operation cannot be assured.

9.2 RESET ROUTINE

The reset routine is the system initialization processing that is executed first.

This routine performs initialization processing that is not related to the memory used by the RX850, i.e., it executes initialization of the target system and sets the pointers (tp, gp, and ep) necessary for program execution.

The reset routine performs the following processing.

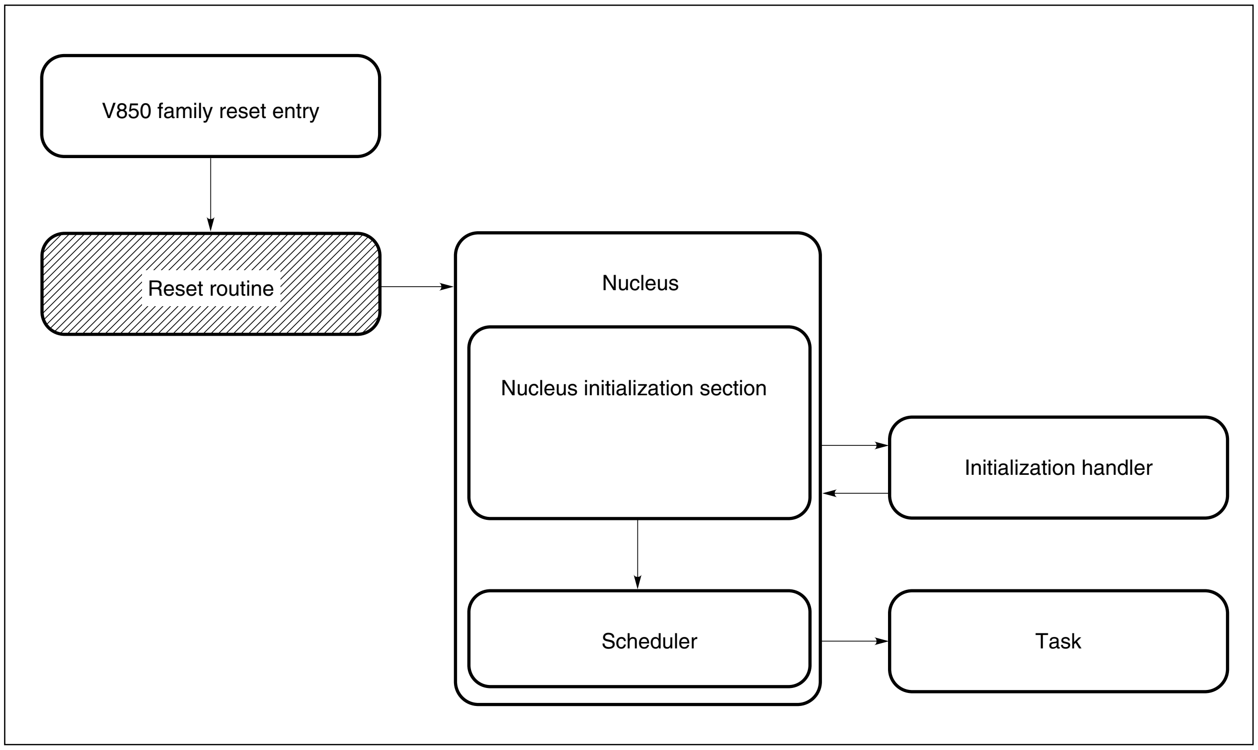
- Setting of tp, gp, and ep registers
- Initialization related to bus control
- Initialization of internal units and peripheral controllers
- Initialization of data area without initial value
- Copy of initialization data
- Branch instruction to nucleus

Because the reset routine must be described in assembly language, make sure that the routine performs the minimum initialization processing and that the remainder of the processing is performed by an initialization handler that can be described in C.

Rewrite the above processing according to the needs of the user.

Figure 9-2 shows the positioning of the hardware initialization section in the RX850.

Figure 9-2. Positioning of Hardware Initialization Section



9.3 NUCLEUS INITIALIZATION SECTION

The nucleus initialization section is a function called by the reset routine. It generates and initializes the management objects based on the information (such as system information or task information) described in the system information table.

The nucleus initialization section performs the following processing:

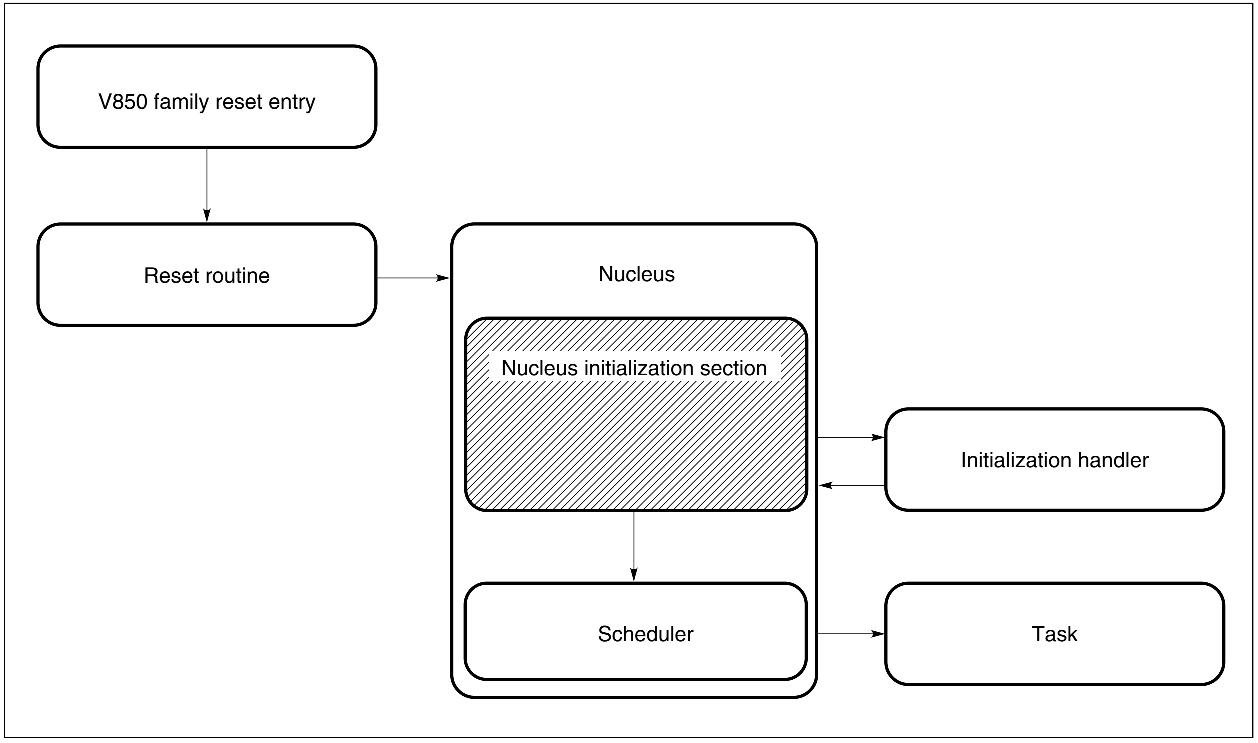
- Generation/initialization of management objects
- Activation of a task[Note]
- Registration of indirectly activated interrupt handlers
- Registration of cyclic handlers
- Registration of time management interrupt handlers (clock handlers)
- Calling of the initialization handler
- Transfer of control to the scheduler

Note The task activated in the nucleus initialization section specifies `ready` as the initial status when a task is created by using the CF definition file. For details, refer to **RX850 User's Manual, Installation**.

Caution The nucleus initialization section is one of the functions provided by the RX850. The user need not code the nucleus initialization section.

Figure 9-3 shows the positioning of the nucleus initialization section in the RX850.

Figure 9-3. Positioning of Nucleus Initialization Section



9.4 INITIALIZATION HANDLER

The initialization handler is a function that is called from the nucleus. The nucleus is initialized after the reset routine has issued a branch instruction to the nucleus. When initialization of the nucleus has been completed, the initialization handler is called. When processing of the initialization handler is completed, the scheduler of the RX850 is activated, and the operation of the OS is started.

Make sure that processing that has not been performed by the reset routine is performed by this initialization handler (see **Section 9.2**). Examples of the processing are as follows:

- Initialization of interrupt control unit
- Initialization of timer/counter unit
- Initialization of system clock
- Branch instruction to nucleus

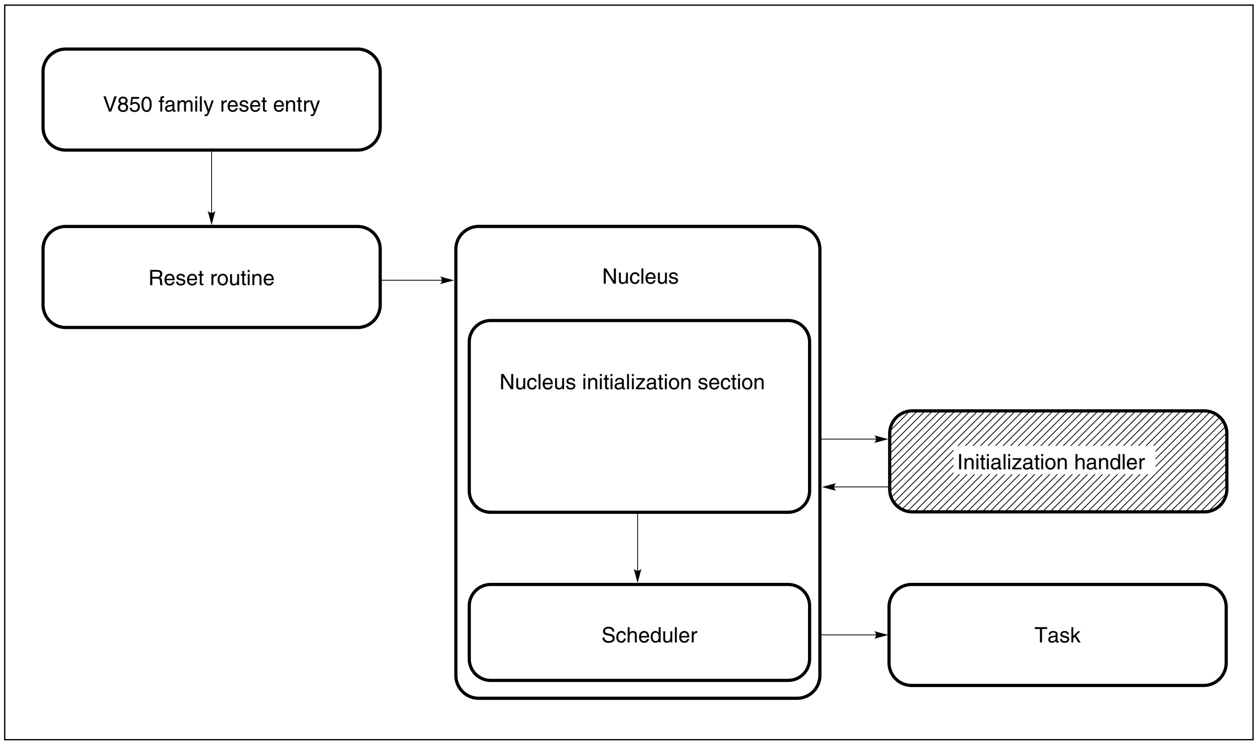
In addition to the above processing, the initialization handler can also initialize a data area without an initial value and copy data with an initial value.

The above processing should be rewritten according to the needs of the user.

Caution In the initialization handler, system calls that can be issued by the interrupt handler and cyclic handler can be issued.

Figure 9-4 shows the positioning of the initialization handler in the RX850.

Figure 9-4. Positioning of Initialization Handler



CHAPTER 10 SYSTEM CALLS

This chapter describes the system calls supported by the RX850.

10.1 OVERVIEW

The user can use system calls to indirectly manipulate those resources that are managed directly by the RX850 (such as counters and queues).

The RX850 supports 62 system calls based on the μITRON 3.0 specifications. System calls can be classified into the following seven groups, according to their functions.

(1) Task management system calls (10)

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

This group provides functions for activating and terminating a task, a function for inhibiting and restarting dispatch processing, a function for changing the task priority, a function to rotating a task ready queue, a function for forcibly releasing a task from the `wait` state, and a function for referencing the task status.

`sta_tsk`	`ext_tsk`	`ter_tsk`	`dis_dsp`	`ena_dsp`
`chg_pri`	`rot_rdq`	`rel_wai`	`get_tid`	`ref_tsk`

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

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 task in the `suspend` state, 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.

`sus_tsk`	`rsm_tsk`	`frsm_tsk`	`slp_tsk`	`tslp_tsk`
`wup_tsk`	`can_wup`			

(3) Synchronous communication system calls (22)

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 event flags or 1-bit event flags, and a function for manipulating mailboxes.

`sig_sem`	`wai_sem`	`preq_sem`	`twai_sem`	`ref_sem`
`set_flg`	`clr_flg`	`wai_flg`	`pol_flg`	`twai_flg`
`ref_flg`	`vset_flg1`	`vclr_flg1`	`vwai_flg1`	`vpol_flg1`
`vtwai_flg1`	`vref_flg1`	`snd_msg`	`rcv_msg`	`prcv_msg`
`trcv_msg`	`ref_mbx`			

(4) Interrupt management system calls (8)

These system calls perform processing that is dependent on the interrupts.

This group provides a function for returning from a directly activated interrupt handler, a function for inhibiting and resuming maskable interrupt acceptance, and a function for changing and referencing the contents of an interrupt control register.

```
ret_int      ret_wup      loc_cpu      unl_cpu      dis_int
ena_int      chg_icr      ref_icr
```

(5) Memory pool management system calls (10)

These system calls allocate memory blocks.

This group provides a function for getting and releasing a memory block and a function for referencing the status of a memory pool.

```
get_blf      pget_blf     tget_blf     rel_blf      ref_mpf
get_blk      pget_blk     tget_blk     rel_blk      ref_mpl
```

(6) Time management system calls (3)

These system calls perform processing that is dependent on time.

This group provides a function for placing a task in the timeout wait state, a function for controlling the state of a cyclic handler, a function for returning from a cyclic handler, and a function for referencing the status of cyclic handler.

```
dly_tsk      act_cyc      ref_cyc
```

(7) System management system calls (2)

These system calls perform processing that varies with the system.

This group provides a function for obtaining version information and a function for referencing the system status.

```
get_ver      ref_sys
```

10.2 CALLING SYSTEM CALLS

System calls issued from tasks or handlers (interrupt handlers or cyclic handlers) written in C are called as C functions. Their parameters are passed as arguments.

When issuing system calls from tasks or handlers written in assembly language, set parameters and a return address according to the function calling rules of the C compiler (CA850 or CCV850), used before calling them with the `jarl` instruction.

Caution The RX850 declares the prototype of a system call in the `stdrx850.h` file. Accordingly, when issuing a system call from a task or handler, the following must be coded to include the header file:

```
#include <stdrx850.h>
```

10.3 DATA TYPES OF PARAMETERS

The system calls supported by the RX850 have parameters that are defined based on data types that conform to the μITRON 3.0 specifications.

Table 10-1 lists the data types of the parameters specified upon the issue of a system call.

Data type macros are defined in the `type.h` header file.

Table 10-1. Data Types

Macro	Data type	Description
B	char	Signed 8-bit integer
H	short	Signed 16-bit integer
INT	int	Signed 32-bit integer
W	long	Signed 32-bit integer
UB	unsigned char	Unsigned 8-bit integer
UH	unsigned short	Unsigned 16-bit integer
UINT	unsigned int	Unsigned 32-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	Start address of processing program
BOOL	char	Boolean value
ID	char	ID number of object
BOOL_ID	char	Existence of waiting task
HNO	char	Specification number of cyclic handler
ER	long	Error code
PRI	char	Priority of task or message
TMO	long	Wait time
CYCTIME	long	Cyclically activated time interval (residual time)
DLYTIME	long	Delay time

10.4 SYSTEM CALL RETURN VALUES

The system call return values supported by the RX850 are based on the μITRON 3.0 specifications.
Table 10-2 lists the system call return values.
Return value macros are defined in the `errno.h` header file.

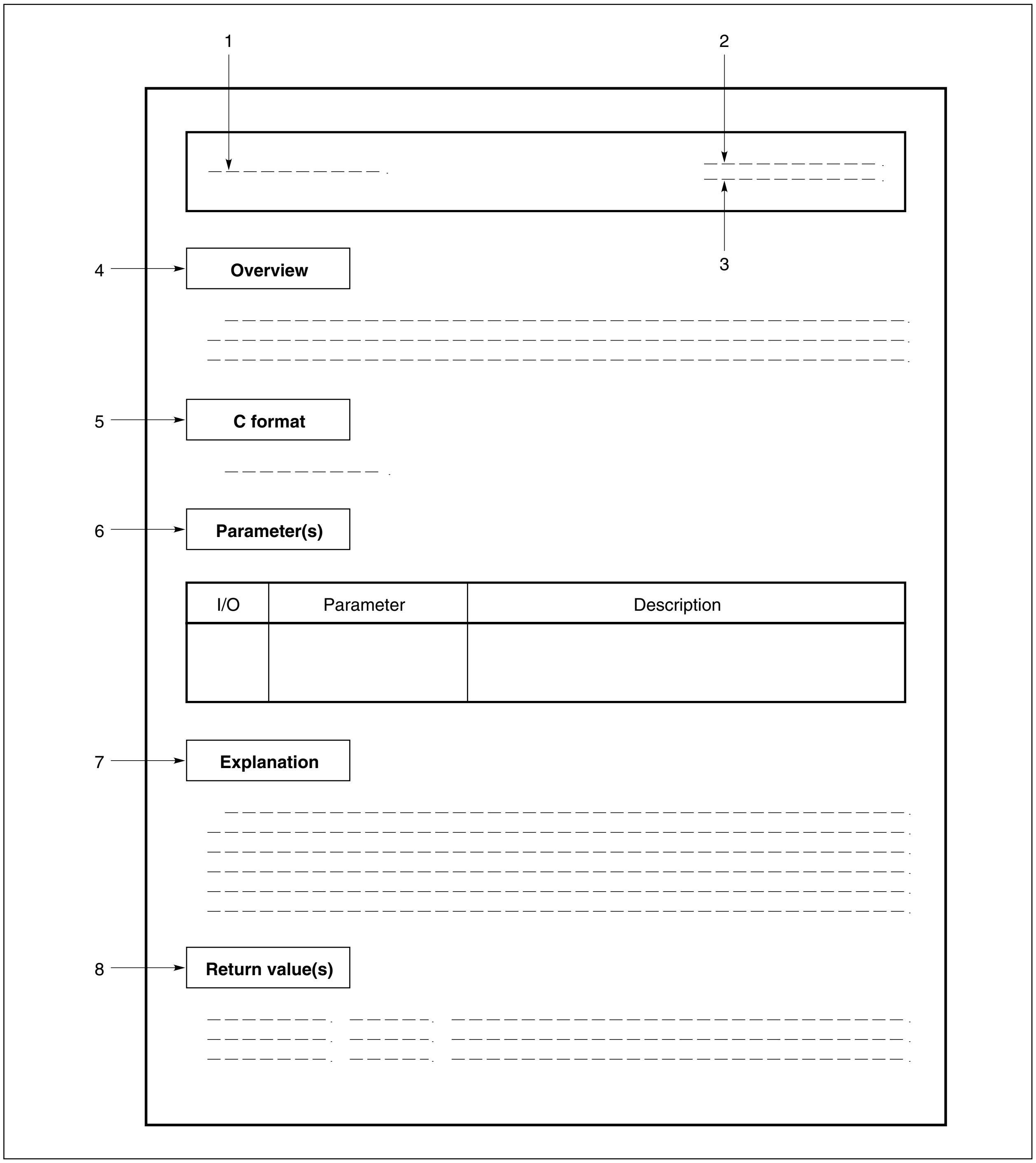
Table 10-2. Return Values

Macro	Value	Description
`E_OK`	`0`	Normal termination
`E_OBJ`	`-63`	The status of the specified object is invalid.
`E_CTX`	`-69`	Context error
`E_QOVR`	`-73`	Count overflow
`E_TMOUT`	`-85`	Timeout/polling failure
`E_RLWAI`	`-86`	The `wait` state was forcibly deleted.

10.5 EXPLANATION OF SYSTEM CALLS

Sections 10.5.1 through **10.5.7** below explain the system calls issued by the RX850 in the following format.

Figure 10-1. 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 **Origin of system call**

Indicates where the system call can be issued.

Task/handler	:	The system call can be issued from both a task and handler (directly activated interrupt handler, indirectly activated interrupt handler, and cyclic handler).
Task	:	The system call can be issued only from a task.
Directly activated interrupt handler	:	The system call can be issued only from a directly activated interrupt handler.
Cyclic handler	:	The system call can be issued only from a cyclic handler.

4 **Overview**

Outlines the functions of the system call.

5 **C format**

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

6 **Parameter(s)**

System call parameters are explained in the following format.
Macros are defined in the `usr.h` and `option.h` header files.

I/O	Parameter	Description
A	B	C

A: Parameter classification

`I` ... Parameter input to the RX850

`O` ... Parameter output from the RX850

B: Parameter data type

C: Description of parameter

7 **Explanation**

Explains the function of a system call.

8 **Return value(s)**

Indicates a system call's return value using a macro and value.
Return value macros are defined in the `errno.h` header file.

10.5.1 Task Management System Calls

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

Table 10-3 lists the task management system calls.

Table 10-3. List of Task Management System Calls

System call	Function	schdl[Note]
`sta_tsk`	Activates another task.	o
`ext_tsk`	Terminates the task which issued the system call.	o
`ter_tsk`	Forcibly terminates another task.	x
`dis_dsp`	Disables dispatch processing.	x
`ena_dsp`	Resumes dispatch processing.	o
`chg_pri`	Change the priority of a task.	o
`rot_rdq`	Rotates a task ready queue.	o
`rel_wai`	Forcibly releases another task from `wait` state.	o
`get_tid`	Acquires a task ID number.	x
`ref_tsk`	Acquires task information.	x

Note schdl indicates whether the scheduler is to be activated.
Whether control is to be passed to another task as a result of scheduler activation, however, depends on the current state.
o: Activates the scheduler.
x: Does not activate the scheduler.

`sta_tsk`

Start Task

Task/handler

Overview

Activates another task.

C format

ER 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 assigns a task execution right to the task specified in *tskid*, then switches that task from the `dormant` state to the `ready` state.

When this system call is issued, if the task cannot acquire a task execution right from the relevant task execution right group, the task itself is queued at the end of the queue of this task execution right group. Thus, the task leaves the `dormant` state and enters the `wait` state (the task execution right wait state).

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.

Cautions 1. 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.

2. The task execution wait state is not canceled by the `rel_wai` system call.

3. Tasks are queued into the queue of the relevant task execution right group according to their priorities.

Return values

E_OK	0	Normal termination
E_OBJ	-63	The specified task is not in the `dormant` state.

ext_tsk

Exit Task

Task

Overview

Terminates the task which issued the system call.

C format

```
void ext_tsk();
```

Parameters

None.

Explanation

This system call returns the task execution right of the task which issued the system call, then switches that task from the `run` state to the `dormant` state.

When this system call is issued, if any tasks are queued into the queue of the relevant task execution right group, the task execution right is not returned, but is instead passed to the first task in the queue.

Thus, the first task is removed from the queue, such that it leaves the `wait` state (the task execution right wait state) and enters the `ready` state.

Cautions 1. If this system call is issued from a handler or in the dispatch inhibited state, its operation is not guaranteed.

2. This system call does not release resources (semaphore count, memory block, etc.) that were acquired before the termination of the task which issued this system call. Accordingly, the user is responsible for releasing those resources before issuing this system call.

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

```
jr    _ext_tsk
```

Return value

None.

`ter_tsk`

Terminate Task

Task

Overview

Forcibly terminates another task.

C format

`ER ter_tsk(ID` *tskid*`);`

Parameters

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

Explanation

This system call returns the task execution right of the task specified in *tskid*, then forcibly switches that task to the `dormant` state.
When this system call is issued, if any tasks are queued into the queue of the relevant task execution right group, the task execution right is not returned, but is passed to the first task in the queue.
Thus, the first task is removed from the queue, such that it leaves the `wait` state (the task execution right wait state) and enters the `ready` state.

Caution This system call does not release those resources (semaphore count, memory block, etc.) that were acquired before the termination of the specified task. Accordingly, the user is responsible for releasing such resources before issuing this system call.

Return values

`E_OK`	`0`	Normal termination
`E_OBJ`	`-63`	The specified task is the task which issued this system call, or the task is in the `dormant` state.

dis_dsp

Disable Dispatch

Task

Overview

Disables dispatch processing.

C format

```
ER dis_dsp();
```

Parameters

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 this system call is issued but before the `ena_dsp` system call is issued, the RX850 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 this 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. After this system call is issued but before the `ena_dsp` system call is issued, if a system call which may cause state transition of the task (such as `ext_tsk` or `wai_sem`) is issued, its operation cannot be assured.

Return values

E_OK	0	Normal termination
E_CTX	-69	This system call was issued after the `loc_cpu` system call was issued.

ena_dsp

Enable Dispatch

Task

Overview

Resumes dispatch processing.

C format

```
ER ena_dsp();
```

Parameters

None.

Explanation

This system call resumes the dispatch processing (task scheduling) that was previously disabled by the issue of a `dis_dsp` system call.
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 this system call is issued, the RX850 merely performs operations on a queue and delays actual scheduling until this 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 this 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 values

`E_OK`	`0`	Normal termination
`E_CTX`	`-69`	This system call was issued after the `loc_cpu` system call had been issued.

`chg_pri`

Change Task Priority

Task/handler

Overview

Changes the priority of a task.

C format

`ER chg_pri(ID` *tskid*, `PRI` *tskpri*`);`

Parameters

I/O	Parameter	Description
`I`	`ID` *tskid*;	Task ID number
`I`	`PRI` *tskpri*;	Task priority

Explanation

This system call changes the value of the task priority specified in *tskid* to that specified in *tskpri*.
If the specified task is in the `run` or `ready` state, this system call rechains the task to the tail of a ready queue according to the priority and also performs priority change processing.

Cautions 1. The value specified for *tskpri* remains effective until this system call is reissued or until the specified task changes to the `dormant` state.

2. The task priority in the RX850 becomes higher as its value decreases.

Return values

`E_OK`	`0`	Normal termination
`E_OBJ`	`-63`	The specified task is not in the `dormant` state.

`rot_rdq`

Rotate Ready Queue

Task/handler

Overview

Rotates a task ready queue.

C format

`ER rot_rdq(PRI` *tskpri*`);`

Parameters

I/O	Parameter	Description
`I`	`PRI` *tskpri*;	Task priority

Explanation

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

Cautions 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 this system call at regular intervals, round-robin scheduling can be achieved.

Return value

`E_OK`	`0`	Normal termination

rel_wai

Release Wait

Task/handler

Overview

Forcibly releases another task from the `wait` state.

C format

```
ER rel_wai(ID tskid);
```

Parameters

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 this 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.

Cautions 1. This system call does not queue release requests. If the specified task is in neither the `wait` nor `wait_suspend` state, `E_OBJ` is returned as the return value.

2. This system call does not release the task execution right wait state. If the specified task is in the task execution right wait state, `E_OBJ` is returned as the return value.

Return values

E_OK	0	Normal termination
E_OBJ	-63	The specified task is in neither the `wait` nor `wait_suspend` state.

get_tid

Get Task Identifier

Task/handler

Overview

Acquires a task ID number.

C format

```
ER get_tid(ID *p_tskid);
```

Parameters

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 handler, FALSE (0) is stored in the area specified in *p_tskid*.

Return value

E_OK	0	Normal termination

ref_tsk

Refer Task Status

Task/handler

Overview

Acquires task information.

C format

```
ER 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

Structure of task information `T_RTSK`

```
typedef struct t_rtsk {
        VP      exinf;      /*  Extended information            */
        PRI     tskpri;     /*  Current priority                */
        UINT    tsksts;     /*  Task status                     */
        UINT    tskwait;    /*  Wait cause                      */
        ID      wid;        /*  ID number of specified object   */
} 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

`tsksts` ... Task state

`TTS_RUN(H'01)`: `run` state

`TTS_RDY(H'02)`: `ready` state

`TTS_WAI(H'04)`: `wait` state

- Wake-up wait state
- Timeout wait state
- Event flag wait state
- Resource wait state
- Message wait state
- Fixed-size memory block wait state
- Variable-size memory block wait state
- 1-bit event flag wait state

`TTS_SUS(H'08)` :	`suspend` state
`TTS_WAS(H'0c)` :	`wait_suspend` state • Combination of the wake-up wait and `suspend` states • Combination of the timeout wait and `suspend` states • Combination of the event flag wait and `suspend` states • Combination of the resource wait and `suspend` states • Combination of the message wait and `suspend` states • Combination of the fixed-size memory block wait and `suspend` states • Combination of the variable-size memory block wait and `suspend` states • Combination of the 1-bit event flag wait and `suspend` states
`TTS_DMT(H'10)` :	`dormant` state
`TTS_WTX(H'20)` :	`wait` state • Task execution right wait state
`TTS_WTS(H'28)` :	`wait_suspend` state • Combination of the task execution right wait and `suspend` states

`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)`	: Variable-size memory block wait state
`TTW_MPF(H'2000)`	: Fixed-size memory block wait state
`TTW_1FLG(H'4000)`	: 1-bit event flag wait state

`wid` ... ID number of specified object (semaphore, event flag, etc.)

Cautions 1. When the value of `tskst s` is other than `TTS_WAI` or `TTS_WAS`, the contents of `tskwait` will be undefined.

2. When the value of `tsksts` is other than `TTS_WTX`, or when the value of `tskwait` is other than `TTW_FLG`, `TTW_SEM`, `TTW_MBX`, `TTW_MPL`, `TTW_MPF`, or `TTW_1FLG`, the contents of `wid` will be undefined.

Return value

`E_OK`	`0`	Normal termination

10.5.2 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 10-4 lists the task-associated synchronization system calls.

Table 10-4. List of Task-Associated Synchronization System Calls

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

Note schdl indicates whether the scheduler is to be activated.
Whether control is to be passed to another task as a result of scheduler activation, however, depends on the current state.
o: Activates the scheduler.
x: Does not activate the scheduler.

sus_tsk

Suspend Task

Task/handler

Overview

Places another task in the `suspend` state.

C format

`ER sus_tsk(ID` *tskid*`);`

Parameters

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

Explanation

This system call issues a suspend request to the task specified in *tskid* (`0x1` is added to the suspend request counter).

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 the RX850 consists of seven bits. Therefore, once the number of suspend requests exceeds 127, this system call returns `E_QOVR` as a return value without incrementing the suspend request counter.

Return values

`E_OK`	`0`	Normal termination
`E_OBJ`	`-63`	The specified task is the task which issued this system call, or the task is in the `dormant` state.
`E_QOVR`	`-73`	The number of suspend requests exceeded 127.

`rsm_tsk`

Resume Task

Task/handler

Overview

Resumes a task in the `suspend` state.

C format

`ER rsm_tsk(ID` *tskid*`);`

Parameters

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 fall to `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 values

`E_OK`	`0`	Normal termination
`E_OBJ`	`-63`	The specified task is in neither the `suspend` nor `wait_suspend` state.

frsm_tsk

Force Resume Task

Task/handler

Overview

Forcibly resumes a task in the `suspend` state.

C format

`ER frsm_tsk(ID` *tskid*`);`

Parameters

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 `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 in neither the `suspend` nor `wait_suspend` state, this system call returns `E_OBJ` as the return value without setting the suspend request counter.

Return values

`E_OK`	`0`	Normal termination
`E_OBJ`	`-63`	The specified task is in neither the `suspend` nor `wait_suspend` state.

slp_tsk

Sleep Task

Task

Overview

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

C format

```
ER slp_tsk();
```

Parameters

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 values

`E_OK`	`0`	Normal termination
`E_RLWAI`	`-86`	The wake-up wait state was forcibly released by the `rel_wai` system call.

`tslp_tsk`

Sleep Task with Timeout

Task

Overview

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

C format

`ER tslp_tsk(TMO` *tmout*`);`

Parameters

I/O	Parameter	Description
`I`	`TMO` *tmout*;	Wait time (unit: basic clock cycle) `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).

The wake-up wait state is released when the wait time specified in *tmout* has elapsed or when the `wup_tsk`, `ret_wup`, or `rel_wai` system call is issued. The task changes from the wake-up wait state to the `ready` state.

Caution When this system call is issued, if `0x0` is specified in *tmout* as the wait time, its operation cannot be assured.

Return values

`E_OK`	`0`	Normal termination
`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.

`wup_tsk`

Wakeup Task

Task/handler

Overview

Wakes up another task.

C format

`ER wup_tsk(ID` *tskid*`);`

Parameters

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* (increments the wake-up request counter 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 the RX850 consists of seven bits. Therefore, when the number of wake-up requests exceeds 127, this system call returns `E_QOVR` as the return value without incrementing the wake-up request counter.

Return values

`E_OK`	`0`	Normal termination
`E_OBJ`	`-63`	The specified task is the task which issued this system call, or the task is in the `dormant` state.
`E_QOVR`	`-73`	The number of wake-up requests exceeded 127.

`can_wup`

Cancel Wakeup Task

Task/handler

Overview

Cancels a request to wake up a task.

C format

`ER can_wup(INT` **p_wupcnt*, `ID` *tskid*`);`

Parameters

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

Explanation

This system call cancels all the wake-up requests issued to the task specified in *tskid* (sets the wake-up request counter to `0x0`).

The number of wake-up requests canceled by this system call is stored in the area specified in *p_wupcnt*.

Return values

`E_OK`	`0`	Normal termination
`E_OBJ`	`-63`	The specified task is in the `dormant` state.

10.5.3 Synchronous Communication System Calls

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

Table 10-5 lists the synchronous communication system calls.

Table 10-5. List of Synchronous Communication System Calls

System call	Function	schdl[Note]
`sig_sem`	Returns resources.	o
`wai_sem`	Acquires resources.	o
`preq_sem`	Acquires resources (polling).	x
`twai_sem`	Acquires resources (with timeout).	o
`ref_sem`	Acquires semaphore information.	x
`set_flg`	Sets a bit pattern.	o
`clr_flg`	Clears a bit pattern.	x
`wai_flg`	Checks a bit pattern.	o
`pol_flg`	Checks a bit pattern (polling).	x
`twai_flg`	Checks a bit pattern (with timeout).	o
`ref_flg`	Acquires event flag information.	x
`vset_flg1`	Sets a bit.	o
`vclr_flg1`	Clears a bit.	x
`vwai_flg1`	Checks a bit.	o
`vpol_flg1`	Checks a bit (polling).	x
`vtwai_flg1`	Checks a bit (with timeout).	o
`vref_flg1`	Acquires 1-bit event flag information.	x
`snd_msg`	Sends a message.	o
`rcv_msg`	Receives a message.	o
`prcv_msg`	Receives a message (polling).	x
`trcv_msg`	Receives a message (with timeout).	o
`ref_mbx`	Acquires mailbox information.	x

Note schdl indicates whether the scheduler is to be activated.
Whether control is to be passed to another task as a result of scheduler activation, however, depends on the current state.
o: Activates the scheduler.
x: Does not activate the scheduler.

sig_sem

Signal Semaphore

Task/handler

Overview

Returns resources.

C format

`ER sig_sem(ID` *semid*`);`

Parameters

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 A semaphore counter managed by the RX850 consists of seven bits. Therefore, when the number of resources exceeds 127, this system call returns `E_QOVR` as its return value without incrementing the semaphore counter.

Return values

E_OK	0	Normal termination
E_QOVR	-73	The resource count exceeded 127.

`wai_sem`

Wait on Semaphore

Task

Overview

Acquires resources.

C format

`ER wai_sem(ID` *semid*`);`

Parameters

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 at the end of 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` or `rel_wai` system call, at which time it changes to the `ready` state.

Return values

`E_OK`	`0`	Normal termination
`E_RLWAI`	`-86`	The resource wait state was forcibly released by a `rel_wai` system call.

preq_sem	**Poll and Request Semaphore**
	Task/handler

Overview

Acquires resources (polling).

C format

```
ER preq_sem(ID semid);
```

Parameters

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 values

`E_OK`	`0`	Normal termination
`E_TMOUT`	`-85`	The resource count for the specified semaphore is `0x0`.

`twai_sem`

Wait on Semaphore with Timeout

Task

Overview

Acquires resources (with timeout).

C format

`ER twai_sem(ID` *semid*, `TMO` *tmout*`);`

Parameters

I/O	Parameter	Description
`I`	`ID` *semid*;	Semaphore ID number
`I`	`TMO` *tmout*;	Wait time (clock interrupt cycles) `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 at the end of 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` or `rel_wai` system call is issued, at which time it changes to the `ready` state.

Return values

`E_OK`	`0`	Normal termination
`E_TMOUT`	`-85`	Wait time elapsed.
`E_RLWAI`	`-86`	The resource wait state was forcibly released by the issue of a `rel_wai` system call.

ref_sem

Refer Semaphore Status

Task/handler

Overview

Acquires semaphore information.

C format

```
ER 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      */
} 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
```

Return value

E_OK	0	Normal termination

`set_flg`

Set Event Flag

Task/handler

Overview

Sets a bit pattern.

C format

`ER 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 bits wide)

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.

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.

Example When this system call is issued, if the bit pattern of the specified event flag is `B'1100` and that specified in *setptn* is `B'1010`, the bit pattern of the specified event flag will be `B'1110`.

Return value

`E_OK`	`0`	Normal termination

`clr_flg`

Clear Event Flag

Task/handler

Overview

Clears a bit pattern.

C format

`ER clr_flg(ID` *flgid*, `UINT` *clrptn*`);`

Parameters

I/O	Parameter	Description
`I`	`ID` *flgid*;	Event flag ID number
`I`	`UINT` *clrptn*;	Bit pattern to clear (32 bits wide)

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.

Example When this system call is issued, if the bit pattern of specified event flag is `B'1100` and that specified in *clrptn* is `B'1010`, the bit pattern of the specified event flag will be `B'1000`.

Return value

`E_OK`	`0`	Normal termination

`wai_flg`

Wait Event Flag

Task

Overview

Checks a bit pattern.

C format

`ER 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 bits wide)
`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 in 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 `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. The RX850 specifies that only one task can be queued into the queue of an event flag. For this reason, if this system call is issued for the event flag for which a waiting task is 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 issuing a `rel_wai` system call, the contents of the area specified in *p_flgptn* will be undefined.

Return values

`E_OK`	`0`	Normal termination
`E_OBJ`	`-63`	This system call was issued for the event flag in which waiting tasks were already queued.
`E_RLWAI`	`-86`	The event flag wait state was forcibly released by a `rel_wai` system call.

pol_flg

Poll Event Flag

Task/handler

Overview

Checks a bit pattern (polling).

C format

```
ER  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 bits wide)
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 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 The RX850 specifies that only one task can be queued into the queue of an event flag. For this reason, if this system call is issued for an event flag in which a waiting task is already queued, this system call returns `E_OBJ` as the return value without performing bit pattern checking.

Return values

`E_OK`	`0`	Normal termination
`E_OBJ`	`-63`	This system call was issued for the event flag in which waiting tasks are already queued.
`E_TMOUT`	`-85`	The bit pattern of the specified event flag does not satisfy the wait condition.

twai_flg

Wait Event Flag with Timeout

Task

Overview

Checks a bit pattern (with timeout).

C format

`ER 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 bits wide)
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 (clock interrupt 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 in 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 `rel_wai` system call is issued, at which time it changes 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. The RX850 specifies that only one task can be queued into the queue of an event flag.
For this reason, if this system call is issued for an event flag in which a waiting task is 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 `rel_wai` system call, the contents of the area specified in *p_flgptn* will be undefined.

Return values

`E_OK`	`0`	Normal termination
`E_OBJ`	`-63`	This system call was issued for the event flag in which waiting tasks were already queued.
`E_TMOUT`	`-85`	Wait time elapsed.
`E_RLWAI`	`-86`	The event flag wait state was forcibly released by the issue of a `rel_wai` system call.

`ref_flg`

Refer Event Flag Status

Task/handler

Overview

Acquires event flag information.

C format

`ER 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 task in queue

`flgptn` ... Current bit pattern

Return value

`E_OK`	`0`	Normal termination

`vset_flg1`

Set 1Bit Event Flag

Task/handler

Overview

Sets a bit.

C format

`ER vset_flg1(ID` *flgid*`);`

Parameters

I/O	Parameter	Description
`I`	`ID` *flgid*;	1-bit event flag ID number

Explanation

This system call sets `1` in the 1-bit event flag specified in *flgid*.

When this system call is issued, if any tasks are queued in the queue of the specified 1-bit event flag, the first task to the task which specifies bit clearing are removed from the queue.

Consequently, the relevant tasks change from the `wait` state (1-bit 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

`vclr_flg1`

Clear 1Bit Event Flag

Task/handler

Overview

Clears a bit.

C format

```
ER  vclr_flg1(ID flgid);
```

Parameters

I/O	Parameter	Description
`I`	`ID` *flgid*;	1-bit event flag ID number

Explanation

This system call sets `0` in the 1-bit event flag specified in *flgid*.

Caution This system call does not queue clear requests. If the 1-bit event flag specified in the current `vclr_flg1` system call has already been cleared by the previous `vclr_flg1` system call, no processing is performed and it is not handled as an error.

Return value

`E_OK`	`0`	Normal termination

`vwai_flg1`

Wait 1Bit Event Flag

Task

Overview

Checks a bit.

C format

`ER  vwai_flg1(ID` *flgid*, `UINT` *wfmode*`);`

Parameters

I/O	Parameter	Description
`I`	`ID` *flgid*;	1-bit event flag ID number
`I`	`UINT` *wfmode*;	Condition satisfaction `TWF_NCL(0)` : Bit is not cleared. `TWF_CLR(1)` : Bit is cleared.

Explanation

This system call checks whether `1` is set in the 1-bit event flag specified in *flgid*.
When this system call is issued, if `1` is not set in the specified 1-bit event flag, this system call queues the task at the end of the queue for the specified 1-bit event flag, then changes it from the `run` state to the `wait` state (1-bit event flag wait state).
The 1-bit event flag wait state is released when the `vset_flg1` or `rel_wai` system call is issued, at which time it changes to the `ready` state.

Return values

`E_OK`	`0`	Normal termination
`E_RLWAI`	`-86`	The 1-bit event flag wait state was forcibly released by a `rel_wai` system call.

vpol_flg1

Poll 1Bit Event Flag

Task/handler

Overview

Checks a bit (polling).

C format

```
ER  vpol_flg1(ID flgid, UINT wfmode);
```

Parameters

I/O	Parameter	Description
I	`ID` *flgid*;	1-bit event flag ID number
I	`UINT` *wfmode*;	Condition satisfaction `TWF_NCL(0)` : Bit is not cleared. `TWF_CLR(1)` : Bit is cleared.

Explanation

This system call checks whether `1` is set in the 1-bit event flag specified in *flgid*.

When this system call is issued, if `1` is not set in the specified 1-bit event flag, this system call returns `E_TMOUT` as the return value.

Return values

`E_OK`	`0`	Normal termination
`E_TMOUT`	`-85`	`1` is not set in the specified 1-bit event flag.

`vtwai_flg1`

Wait 1Bit Event Flag with Timeout

Task

Overview

Checks a bit (with timeout).

C format

`ER vtwai_flg1(ID` *flgid*, `UINT` *wfmode*, `TMO` *tmout*`);`

Parameters

I/O	Parameter	Description
`I`	`ID` *flgid*;	1-bit event flag ID number
`I`	`UINT` *wfmode*;	Condition satisfaction `TWF_NCL(0)` : Bit is not cleared. `TWF_CLR(1)` : Bit is cleared.
`I`	`TMO` *tmount*;	Wait time (clock interrupt cycles) `TMO_POL(0)` : Quick return `TMO_FEVR(-1)` : Permanent wait Value : Wait time

Explanation

This system call checks whether `1` is set in the 1-bit event flag specified in *flgid*.

Upon the issue of this system call, if `1` is not set in the specified 1-bit event flag, this system call queues the task at the end of the queue for the specified 1-bit event flag, then changes it from the `run` state to the `wait` state (1-bit event flag wait state).

The 1-bit event flag wait state is released upon the elapse of the wait time specified in *tmout*, or when the `vset_flg1` or `rel_wai` system call is issued, at which time it changes to the `ready` state.

Return values

`E_OK`	`0`	Normal termination
`E_TMOUT`	`-85`	Wait time elapsed.
`E_RLWAI`	`-86`	The 1-bit event flag wait state was forcibly released by the issue of a `rel_wai` system call.

vref_flg1

Refer 1Bit Event Flag Status

Task/handler

Overview

Acquires 1-bit event flag information.

C format

ER vref_flg1(T_RFLG *pk_rflg*, ID *flgid*);

Parameters

I/O	Parameter	Description
O	T_RFLG **pk_rflg*;	Start address of packet used to store 1-bit event flag information
I	ID *flgid*;	1-bit event flag ID number

Structure of 1-bit event flag information T_RFLG

```
typedef struct t_rflg {
        VP      exinf;     /*  Extended information      */
        BOOL_ID wtsk;      /*  Existence of waiting task */
        UINT    flgptn;    /*  Current bit value         */
} T_RFLG;
```

Explanation

This system call stores, in the packet specified in *pk_rflg*, the 1-bit event flag information (extended information, existence of waiting task, etc.) for the 1-bit event flag specified in *flgid*.

1-bit 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 value

Return value

E_OK	0	Normal termination

snd_msg

Send Message to Mailbox

Task/handler

Overview

Sends a message.

C format

`ER snd_msg(ID` *mbxid*, `T_MSG` **pk_msg*`);`

Parameters

I/O	Parameter	Description
I	`ID` *mbxid*;	Mailbox ID number
I	`T_MSG` **pk_msg*;	Address of area 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.

Cautions 1. A message is queued into the message queue of a specified mailbox in the order (FIFO order, priority order) specified when the mailbox was created (at configuration).

2. The RX850 uses the first four 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 this system call.

Return value

`E_OK`	`0`	Normal termination

`rcv_msg`

Receive Message from Mailbox

Task

Overview

Receives a message.

C format

```
ER  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 at the end of 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` or `rel_wai` system call is issued, at which time the state changes from the message wait state to the `ready` state.

Return values

`E_OK`	`0`	Normal termination
`E_RLWAI`	`-86`	The message wait state was forcibly released by a `rel_wai` system call.

prcv_msg

Poll and Receive Message from Mailbox

Task/handler

Overview

Receives a message (polling).

C format

```
ER  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 values

`E_OK`	`0`	Normal termination
`E_TMOUT`	`-85`	No message exists in a specified mailbox.

trcv_msg

Receive Message from Mailbox with Timeout

Task

Overview

Receives a message (with timeout).

C format

`ER 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 (clock interrupt 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 at the end of 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` or `rel_wai` system call is issued, at which time the state changes from the message wait state to the `ready` state.

Return values

`E_OK`	`0`	Normal termination
`E_TMOUT`	`-85`	Wait time elapsed.
`E_RLWAI`	`-86`	The message wait state was forcibly released by a `rel_wai` system call.

ref_mbx

Refer Mailbox Status

Task/handler

Overview

Acquires mailbox information.

C format

```
ER  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   **ppk_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) : There is no waiting task.
              Value    : ID number of first task of queue
ppk_msg ... Existence of waiting message
              NADR(-1) : No waiting message
              Value    : Address of first message of queue
```

Return value

E_OK	0	Normal termination

10.5.4 Interrupt Management System Calls

This section explains a group of system calls (interrupt management system calls) that perform processing that depends on interrupts.

Table 10-6 lists the interrupt management system calls.

Table 10-6. List of Interrupt Management System Calls

System call	Function	schdl[Note]
`ret_int`	Returns from a directly activated interrupt handler.	o
`ret_wup`	Wakes up another task and returns from a directly activated interrupt handler.	o
`loc_cpu`	Disables the acceptance of maskable interrupts and dispatch processing.	x
`unl_cpu`	Resumes the acceptance of maskable interrupts and dispatch processing.	o
`dis_int`	Disables the acceptance of maskable interrupts.	x
`ena_int`	Resumes the acceptance of maskable interrupts.	x
`chg_icr`	Changes the contents of an interrupt control register.	x
`ref_icr`	Acquires the contents of an interrupt control register.	x

Note schdl indicates whether the scheduler is to be activated.
Whether control is to be passed to another task as a result of scheduler activation, however, depends on the current state.
o: Activates the scheduler.
x: Does not activate the scheduler.

`ret_int`

Return from Interrupt Handler

Directly activated interrupt handler

Overview

Returns from a directly activated interrupt handler.

C format

```
void ret_int();
```

Parameters

None.

Explanation

This system call returns from a directly activated interrupt handler.

If a system call (`chg_pri`, `sig_sem`, etc.) requiring task scheduling is issued from a directly activated interrupt handler, the RX850 merely queues the tasks into the queue and delays actual scheduling until a system call (this system call 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.

Cautions 1. This system call does not notify the external interrupt controller of the termination of processing (issue of 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 call.

2. When an indirectly activated interrupt handler is written in C, return from the indirectly activated interrupt handler is coded as follows:

```
return(0xff);
```

3. When an indirectly activated interrupt handler is written in assembly language, return from the indirectly activated interrupt handler is coded as follows:

```
mov    0xff, r10
jmp    [lp]
```

Return value

None.

`ret_wup`

Return and Wakeup Task

Directly activated interrupt handler

Overview

Wakes up another task and returns from a directly activated interrupt handler.

C format

`void  ret_wup(ID` *tskid*`);`

Parameters

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

Explanation

This system call returns from a directly activated 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 directly activated interrupt handler, the RX850 merely queues the tasks into a queue and delays the actual scheduling until a system call (this system call or `ret_int` system call) is issued to return from the directly activated interrupt handler. Then, the queued tasks are all performed at one time.

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 call.

2. This system call is responsible only for return from a directly activated interrupt handler if one of the following errors occurs:

- **A specified task is in the `run` or `dormant` state.**
- **The number of wake-up requests exceeded 127.**

3. When an indirectly activated interrupt handler is written in C, another task is woken up, or a return from the indirectly activated interrupt handler is performed by coding the following:

`return(ID` *tskid*`);`

4. When an indirectly activated interrupt handler is written in the assembly language, code as follows to wake up another task or to return from the indirectly activated interrupt handler:

```
mov    tskid,  r10
jmp    [lp]
```

Return value

None.

`loc_cpu`

Lock CPU

Task

Overview

Disables the acceptance of maskable interrupts and dispatch processing.

C format

```
ER  loc_cpu();
```

Parameters

None.

Explanation

This system call disables the acceptance of maskable interrupts and dispatch processing (task scheduling).
The acceptance of maskable interrupts and dispatch processing is inhibited until the `unl_cpu` system call is issued after the issue of this system call.
If a maskable interrupt occurs after this system call is issued but before the `unl_cpu` system call is issued, the RX850 delays processing for the interrupt (directly activated interrupt handler or indirectly activated interrupt handler) until the `unl_cpu` system call is issued. If a system call (`chg_pri`, `sig_sem`, etc) requiring task scheduling is issued, the RX850 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.

Cautions 1. This system call does not queue disable requests. Accordingly, if this system call has already been issued 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.

2. After this system call is issued but before the `unl_cpu` system call is issued, if a system call which may cause state transition of the task (such as `ext_tsk` or `wai_sem`) is issued, its operation cannot be assured.

3. After this system call is issued but before the `unl_cpu` system call is issued, if the `ena_int` system call is issued, its operation cannot be assured.

Return value

`E_OK`	`0`	Normal termination

unl_cpu

Unlock CPU

Task

Overview

Resumes the acceptance of maskable interrupts and dispatch processing.

C format

```
ER  unl_cpu();
```

Parameters

None.

Explanation

This system call resumes the acceptance of maskable interrupts and dispatch processing (task scheduling) inhibited by the `loc_cpu` system call.
If a maskable interrupt occurs after the `loc_cpu` system call is issued but before this system call is issued, the RX850 delays processing for the interrupt (directly activated interrupt handler or indirectly activated interrupt handler) until this system call is issued. If a system call (`chg_pri`, `sig_sem`, etc) requiring task scheduling is issued, the RX850 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.

Cautions 1. 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.

2. Acceptance of maskable interrupts that was inhibited by the issue of the `dis_int` system call is resumed by this system call.

3. Dispatch processing that was inhibited by the issue of the `dis_dsp` system call is resumed by this system call.

Return value

`E_OK`	`0`	Normal termination

`dis_int`

Disable Interrupt

Task/handler

Overview

Disables the acceptance of maskable interrupts.

C format

```
ER  dis_int();
```

Parameters

None.

Explanation

This system call disables the acceptance of maskable interrupts.

Once this system call has been issued, the acceptance of maskable interrupts is disabled until the `ena_int` system call is issued.

If a maskable interrupt occurs after this system call has been issued but before the `ena_int` system call is issued, the RX850 delays processing for the interrupt (directly activated interrupt handler or indirectly activated interrupt handler) until the `ena_int` system call is issued.

Caution This system call does not queue disable requests. When the current `dis_int` system call is issued, if the acceptance of maskable interrupts has already been disabled by the previous `dis_int` system call, no processing is performed and it is not handled as an error.

Return value

`E_OK`	0	Normal termination

`ena_int`

Enable Interrupt

Task/handler

Overview

Resumes the acceptance of maskable interrupts.

C format

```
ER  ena_int();
```

Parameters

None.

Explanation

This system call resumes the acceptance of maskable interrupts, which has been disabled by the issue of the `dis_int` system call.

If a maskable interrupt occurs after the `dis_int` system call is issued but before the `ena_int` system call is issued, the RX850 delays processing for the interrupt (directly activated interrupt handler or indirectly activated interrupt handler) until the `ena_int` system call is issued.

Caution This system call does not queue resume requests. When the current `ena_int` system call is issued, if the acceptance of maskable interrupts has already been resumed by the previous `ena_int` system call, no processing is performed and it is not handled as an error.

Return value

`E_OK`	`0`	Normal termination

`chg_icr`

Change Interrupt Control Register

Task/handler

Overview

Changes the contents of an interrupt control register.

C format

`ER chg_icr(UINT` *eintno*, `UB` *icrcmd*`);`

Parameters

I/O	Parameter	Description
`I`	`UINT` *eintno*;	Interrupt request number
`I`	`UB` *icrcmd*;	Specification of an interrupt request flag `ICR_CLRINT (0x20)`: No interrupt request is made. Specification of an interrupt mask flag `ICR_CLRMSK (0x10)`: Enables interrupt processing. `ICR_SETMSK (0x40)`: Disables interrupt processing. Specification of the change of an interrupt priority `ICR_CHGLVL (0x08)`: Changes an interrupt priority. Specification of an interrupt priority Value (`0` to `7`) : Interrupt priority.

Explanation

This system call changes the contents of the interrupt control register specified in *eintno* to the value specified in *icrcmd*.

The specification format of *icrcmd* is described below.

- *icrcmd* = `ICR_CLRINT`
  Changes the interrupt request flag of the interrupt control register to `0`.
- *icrcmd* = `ICR_CLRMSK`
  Changes the interrupt mask flag of the interrupt control register to `0`.
- *icrcmd* = `ICR_SETMSK`
  Changes the interrupt mask flag of the interrupt control register to `1`.
- *icrcmd* = `(ICR_CHGLVL|`value`)`
  Changes the interrupt priority of the interrupt control register to the specified value.
  A value of `0` corresponds to level 0, while `7` corresponds to level 7.

Caution For *eintno*, specify the value obtained from the following:
(The exception code of the relevant interrupt request number - `0x80`) / `0x10`

Return value

`E_OK`	`0`	Normal termination

ref_icr

Refer Interrupt Control Register Status

Task/handler

Overview

Acquires the contents of an interrupt control register.

C format

```
ER  ref_icr(UB *p_regptn, UINT eintno);
```

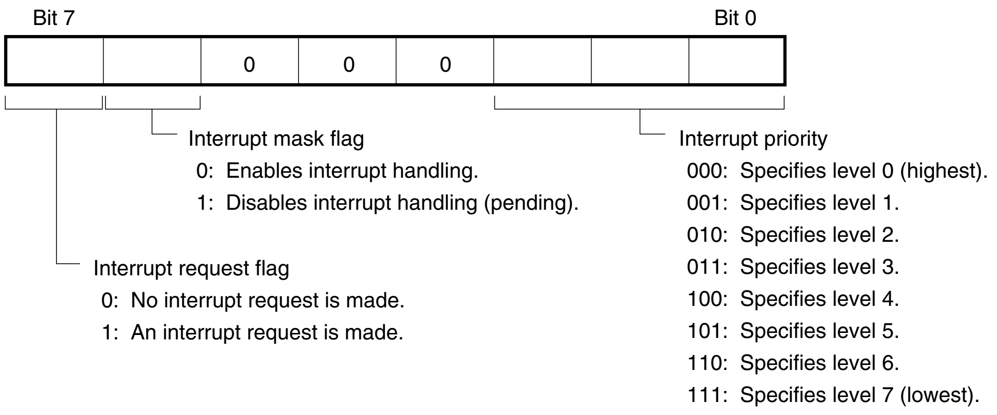
Parameters

I/O	Parameter	Description
O	UB *p_regptn;	Address of the area used to store the contents of an interrupt control register
I	UINT eintno;	Interrupt request number

Explanation

This system call stores, into the area specified in *p_regptn*, the contents of the interrupt control register specified in *eintno*.

The following figure shows the acquired contents of an interrupt control register.



Caution For *eintno*, specify the value obtained from the following:
(The exception code of the relevant interrupt request number - `0x80`) / `0x10`

Return value

E_OK	0	Normal termination

10.5.5 Memory Pool Management System Calls

This section explains a group of system calls that allocate memory blocks (memory pool management system calls).

Table 10-7 lists the memory pool management system calls.

Table 10-7. List of Memory Pool Management System Calls

System call	Function	schdl[Note]
`get_blf`	Acquires a fixed-size memory block.	o
`pget_blf`	Acquires a fixed-size memory block (polling).	x
`tget_blf`	Acquires a fixed-size memory block (with timeout).	o
`rel_blf`	Returns a fixed-size memory block.	o
`ref_mpf`	Acquires fixed-size memory pool information.	x
`get_blk`	Acquires a variable-size memory block.	o
`pget_blk`	Acquires a variable-size memory block (polling).	x
`tget_blk`	Acquires a variable-size memory block (with timeout).	o
`rel_blk`	Returns a variable-size memory block.	o
`ref_mpl`	Acquires variable-size memory pool information.	x

Note schdl indicates whether the scheduler is to be activated.
Whether control is to be passed to another task as a result of scheduler activation, however, depends on the current state.
o: Activates the scheduler.
x: Does not activate the scheduler.

`get_blf`

Get Fixed-size Memory Block

Task

Overview

Acquires a fixed-size memory block.

C format

`ER get_blf(VP *p_blf, ID mpfid);`

Parameters

I/O	Parameter	Description
O	VP *p_blf;*	Address of area used to store the start address of the fixed-size memory block
I	ID *mpfid*;	Fixed-size memory pool ID number

Explanation

This system call acquires a fixed-size memory block from the fixed-size memory pool specified in *mpfid* and stores its start address into the area specified in *p_blf*.

If no fixed-size memory block can be acquired from a specified fixed-size memory pool (when there is no free area) upon the issue of this system call, this system call places the task at the end of the queue of a specified fixed-size memory pool before changing its state from the `run` state to the `wait` state (fixed-size memory block wait state).

The fixed-size memory block wait state is released upon the issue of a `rel_blf` or `rel_wai` system call, at which time it changes to the `ready` state.

Caution NEC recommends that, under the RX850, a memory block be used as an area for messages exchanging between tasks via mailboxes. The first four bytes of a message are used as a link area that is used to queue that message. The RX850, thus, clears only the first four bytes of any acquired fixed-size memory block.

Return values

E_OK	0	Normal termination
E_RLWAI	-86	The fixed-size memory block wait state was forcibly released using a `rel_wai` system call.

`pget_blf`

Poll and Get Fixed-size Memory Block

Task/handler

Overview

Acquires a fixed-size memory block (polling).

C format

`ER pget_blf(VP` **p_blf*, `ID` *mpfid*`);`

Parameters

I/O	Parameter	Description
`O`	`VP` **p_blf*;	Address of area used to store the start address of a fixed-size memory block
`I`	`ID` *mpfid*;	Fixed-size memory pool ID number

Explanation

This system call acquires a fixed-size memory block from the fixed-size memory pool specified in *mpfid* and stores its start address into the area specified in *p_blf*.

When this system call is issued, if no fixed-size memory block can be acquired from the specified fixed-size memory pool (when there is no free area), this system call returns `E_TMOUT` as the return value.

Caution NEC recommends that, under the RX850, a memory block be used as an area for messages exchanging between tasks via mailboxes. The first four bytes of a message are used as a link area that is used to queue that message. The RX850, thus, clears only the first four bytes of any acquired fixed-size memory block.

Return values

`E_OK`	`0`	Normal termination
`E_TMOUT`	`-85`	There is no free space in the specified fixed-size memory pool.

tget_blf

Get Fixed-size Memory Block with Timeout

Task

Overview

Acquires a fixed-size memory block (with timeout).

C format

`ER  tget_blf(VP *p_blf, ID mpfid, TMO tmout);`

Parameters

I/O	Parameter		Description
O	VP	**p_blf*;	Address of area used to store the start address of a fixed-size memory block
I	ID	*mpfid*;	Fixed-size memory pool ID number
I	TMO	*tmout*;	Wait time (clock interrupt cycles) TMO_POL(0) : Quick return TMO_FEVR(-1) : Permanent wait Value : Wait time

Explanation

This system call acquires a fixed-size memory block from the fixed-size memory pool specified in *mpfid* and stores its start address into the area specified in *p_blf*.

If no fixed-size memory block can be acquired from a specified fixed-size memory pool (when there is no free area) when this system call is issued, this system call places the task at the end of the queue of a specified fixed-size memory pool before changing it from the `run` state to the `wait` state (fixed-size memory block wait state).

The fixed-size memory block wait state is released when the wait time specified in *tmout* elapses, or when the `rel_blf` or `rel_wai` system call is issued. Then, the state changes to the `ready` state.

Caution NEC recommends that, under the RX850, a memory block be used as an area for messages exchanging between tasks via mailboxes. The first four bytes of a message are used as a link area that is used to queue that message. The RX850, thus, clears only the first four bytes of any acquired fixed-size memory block.

Return values

E_OK	0	Normal termination
E_TMOUT	-85	Timeout elapsed.
E_RLWAI	-86	The fixed-size memory block wait state was forcibly released by a `rel_wai` system call.

`rel_blf`

Release Fixed-size Memory Block

Task/handler

Overview

Returns a fixed-size memory block.

C format

`ER  rel_blf(ID` *mpfid*, `VP` *blf*`);`

Parameters

I/O	Parameter	Description
`I`	`ID` *mpfid*;	Fixed-size memory pool ID number
`I`	`VP` *blf*;	Start address of fixed-size memory block

Explanation

This system call returns the fixed-size memory block specified in *blf* to the fixed-size memory pool specified in *mpfid*.

When this system call is issued, if any tasks are queued into the queue of the specified fixed-size memory pool, the fixed-size memory block is not returned, but is passed to the first task in the queue.

Consequently, the relevant task is removed from the queue, and changes from the `wait` state (fixed-size memory block wait state) to the `ready` state, or from the `wait_suspend` state, to the `suspend` state.

Cautions 1. RX850 does not clear the contents of memory when returning a fixed-size memory block. Accordingly, the contents of an acquired fixed-size memory block will be undefined.

2. The fixed-size memory block to be returned must be the same as that specified upon the issue a `get_blf`, `pget_blf`, or `tget_blf` system call.

Return value

`E_OK`	`0`	Normal termination

ref_mpf

Refer Fixed-size Memory Pool Status

Task/handler

Overview

Acquires a fixed-size memory pool information.

C format

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

Parameters

I/O	Parameter	Description
O	`T_RMPF` **pk_rmpf*;	Start address of packet used to store fixed-size memory pool information
I	`ID` *mpfid*;	Fixed-size memory pool ID number

Structure of fixed-size memory pool information `T_RMPF` structure

```
typedef struct  t_rmpf {
        VP      exinf;     /*  Extended information      */
        BOOL_ID wtsk;      /*  Existence of waiting task */
        INT     frbcnt;    /*  Number of free blocks     */
} T_RMPF;
```

Explanation

This system call stores the fixed-size memory pool information (extended information, existence of waiting tasks, etc.) for the fixed-size memory pool specified in *mpfid* into the packet specified in *pk_rmpf*.

Fixed-size memory pool 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 the queue

`frbcnt` ... Number of free blocks

Return value

`E_OK`	0	Normal termination

get_blk

Get Variable-size Memory Block

Task

Overview

Acquires a variable-size memory block.

C format

```
ER  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 variable-size memory block
I	ID mplid;	Variable-size memory pool ID number
I	INT blksz;	Variable-size memory block size (number of bytes)

Explanation

This system call acquires a variable-size memory block of the size specified by *blksz*[Note] from the variable-size memory pool specified by *mplid*, and stores the first address of the acquired memory block into the area specified by *p_blk*.

Because a 4-byte management area is appended to a variable-size memory block, the size of the memory block that can be acquired is less than the size of the variable-size memory pool.

If no variable-size memory block can be acquired from a specified variable-size memory pool (when there is no free area) upon the issue of this system call, this system call places the task at the end of the queue of a specified variable-size memory pool before changing its state from the `run` state to the `wait` state (variable-size memory block wait state).

If a task is queued when this system call is issued, it unconditionally waits for a variable-size memory block, regardless of the size of the vacant variable-size memory block. In other words, the task is queued only on an FIFO basis, and is not queued according to its priority or in the sequence of the required block size.

The variable-size memory block wait state is released upon the issue of a `rel_blk` or `rel_wai` system call, at which time it changes to the `ready` state.

Note The variable-size memory block is acquired in 4-byte units. Therefore, specify a multiple of four for *blksz*. If any other value is specified, the number of variable-size memory blocks actually acquired differs from the specified value of *blksz* because the memory block is acquired in 4-byte units.

Cautions 1. NEC recommends that, under the RX850, a memory block be used as an area for messages exchanging between tasks via mailboxes. The first four bytes of a message are used as a link area that is used to queue that message. The RX850, thus, clears only the first four bytes of any acquired variable-size memory block.

2. If this system call is issued in the dispatch disabled status or from a handler, the operation is not guaranteed.

Return values

`E_OK`	`0`	Normal termination
`E_RLWAI`	`-86`	The variable-size memory block wait state was forcibly released using a `rel_wai` system call.

`pget_blk`

Poll and Get Variable-size Memory Block

Task/handler

Overview

Acquires a variable-size memory block (polling).

C format

`ER 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 variable-size memory block
I	`ID` *mplid*;	Variable-size memory pool ID number
I	`INT` *blksz*;	Variable-size memory block size (number of bytes)

Explanation

This system call acquires a variable-size memory block of the size specified by *blksz*Note from the variable-size memory pool specified by *mplid*, and stores the first address of the acquired memory block into the area specified by *p_blk*.

Because a 4-byte management area is appended to a variable-size memory block, the size of the memory block that can be acquired is less than the size of the variable-size memory pool.

When this system call is issued, if no variable-size memory block can be acquired from the specified variable-size memory pool (when there is no free area), this system call returns `E_TMOUT` as the return value.

If a task is queued, to await the specified variable-size memory pool, when this system call is issued, polling fails unconditionally. Even if a vacant block of the necessary size exists in the variable-size memory pool, if there is a task waiting to acquire the larger memory block, acquisition of the memory block by that task takes precedence.

Note The variable-size memory block is acquired in 4-byte units. Therefore, specify a multiple of four for *blksz*. If any other value is specified, the number of variable-size memory blocks actually acquired differs from the specified value of *blksz* because the memory block is acquired in 4-byte units.

Caution NEC recommends that, under the RX850, a memory block be used as an area for messages exchanging between tasks via mailboxes. The first four bytes of a message are used as a link area that is used to queue that message. The RX850, thus, clears only the first four bytes of any acquired variable-size memory block.

Return values

`E_OK`	`0`	Normal termination
`E_TMOUT`	`-85`	There is no free space in the specified variable-size memory pool.

`tget_blk`

Get Variable-size Memory Block with Timeout

Task

Overview

Acquires a variable-size memory block (with timeout).

C format

`ER 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 variable-size memory block
`I`	`ID`	*mplid*;	Variable-size memory pool ID number
`I`	`INT`	*blksz*;	Variable-size memory block size (number of bytes)
`I`	`TMO`	*tmout*;	Wait time (clock interrupt cycles) `TMO_POL(0)` : Quick return `TMO_FEVR(-1)` : Permanent wait Value : Wait time

Explanation

This system call acquires a variable-size memory block of the size specified by *blksz*[Note] from the variable-size memory pool specified by *mplid*, and stores the first address of the acquired memory block into the area specified by *p_blk*.
Because a 4-byte management area is appended to a variable-size memory block, the size of the memory block that can be acquired is less than the size of the variable-size memory pool.
If no variable-size memory block can be acquired from a specified variable-size memory pool (when there is no free area) when this system call is issued, this system call places the task at the end of the queue of a specified variable-size memory pool before changing it from the `run` state to the `wait` state (variable-size memory block wait state).
Tasks are queued on an FIFO basis.
If a task is queued when this system call is issued, it unconditionally enters the variable-size memory block wait state, regardless of the size of the vacant variable-size memory block.
The variable-size memory block wait state is released when the wait time specified in *tmout* elapses, or when a `rel_blk` or `rel_wai` system call is issued. Then, the state changes to the `ready` state.

Note The variable-size memory block is acquired in 4-byte units. Therefore, specify a multiple of four for *blksz*. If any other value is specified, the number of variable-size memory blocks actually acquired differs from the specified value of *blksz* because the memory block is acquired in 4-byte units.

Caution NEC recommends that, under the RX850, a memory block be used as an area for messages exchanging between tasks via mailboxes. The first four bytes of a message are used as a link area that is used to queue that message. The RX850, thus, clears only the first four bytes of any acquired variable-size memory block.

Return values

E_OK	0	Normal termination
E_TMOUT	-85	Timeout elapsed.
E_RLWAI	-86	The variable-size memory block wait state was forcibly released by a rel_wai system call.

`rel_blk`

Release Variable-size Memory Block

Task/handler

Overview

Returns a variable-size memory block.

C format

`ER rel_blk(ID` *mplid*`, VP` *blk*`);`

Parameters

I/O	Parameter	Description
`I`	`ID` *mplid*;	Variable-size memory pool ID number
`I`	`VP` *blk*;	Start address of variable-size memory block

Explanation

This system call returns the variable-size memory block specified in *blk* to the variable-size memory pool specified in *mplid*. If a contiguous area of a size satisfying the requirement by the waiting task is acquired in the variable-size memory pool as a result of returning of the variable-size memory block, the task is released from the queue and acquires the variable-size memory block.
When this system call is issued, if any tasks are queued into the queue of the specified variable-size memory pool, the variable-size memory block is not returned, but is passed to the first task in the queue.
Consequently, the relevant task is removed from the queue, and changes from the `wait` state (variable-size memory block wait state) to the `ready` state, or from the `wait_suspend` state, to the `suspend` state.

Cautions 1. The RX850 does not clear the contents of memory when returning a variable-size memory block. Accordingly, the contents of an acquired variable-size memory block will be undefined.

2. The variable-size 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

ref_mpl

Refer Variable-size Memory Pool Status

Task/handler

Overview

Acquires a variable-size memory pool information.

C format

`ER 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 variable-size memory pool information
I	`ID` *mplid*;	Variable-size memory pool ID number

Structure of variable-size 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 vacant area (bytes)  */
        INT     maxsz;    /*  Size of maximum vacant area (bytes) */
} T_RMPL;
```

Explanation

This system call stores the variable-size memory pool information (extended information, existence of waiting tasks, etc.) for the variable-size memory pool specified in *mplid* into the packet specified in *pk_rmpl*.
Variable-size memory pool 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 the queue

`frsz` ... Total size of vacant area (bytes)
`maxsz` … Size of maximum vacant area (bytes)

Return value

`E_OK`	`0`	Normal termination

10.5.6 Time Management System Calls

This section explains a group of system calls (time management system calls) that perform processing that is dependent on time.

Table 10-8 lists the time management system calls.

Table 10-8. List of Time Management System Calls

System call	Function	schdl[Note]
`dly_tsk`	Changes the task to the timeout wait state.	o
`act_cyc`	Controls the activity state of a cyclic handler.	x
`ref_cyc`	Acquires cyclic handler information.	x

Note schdl indicates whether the scheduler is to be activated.
Whether control is to be passed to another task as a result of scheduler activation, however, depends on the current state.
o: Activates the scheduler.
x: Does not activate the scheduler.

`dly_tsk`

Delay Task

Task

Overview

Changes the task to the timeout wait state.

C format

`ER dly_tsk(DLYTIME` *dlytim*`);`

Parameters

I/O	Parameter	Description
`I`	`DLYTIME` *dlytim*;	Delay (clock interrupt 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 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 state changes to the `ready` state.

Caution The timeout wait state is released by neither the `wup_tsk` nor `ret_wup` system call.

Return values

`E_OK`	`0`	Normal termination
`E_RLWAI`	`-86`	The timeout wait state was forcibly released by the issue of a `rel_wai` system call.

act_cyc

Activate Cyclic Handler

Task/handler

Overview

Controls the activity state of a cyclic handler.

C format

```
ER act_cyc(HNO cycno, UINT cycact);
```

Parameters

I/O	Parameter	Description
I	`HNO` *cycno*;	Specification number of cyclic handler
I	`UINT` *cycact*;	Specification of activity state, cycle counter, and queuing `TCY_OFF(0)` : OFF state `TCY_ON(1)` : ON state `TCY_INI(2)` : Initializes the cycle counter. `TCY_ULNK(4)`: Removes the handler from the timer queue.

Explanation

This system call changes the activity state of the cyclic 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 cyclic handler to the OFF state. Even when the activation time is reached, the cyclic handler is not activated.

  Caution Under the RX850, while the cyclic handler is queued in the timer queue, the cycle counter continues to count even when the activity state of that handler is OFF.

- *cycact* = `TCY_ON`
  Changes the activity state of a cyclic handler to the ON state. When the activation time is reached, the specified cyclic handler is activated.
- *cycact* = `TCY_INI`
  Queues the specified cyclic handler into the timer queue, then initializes the cycle counter.

  Caution When this system call is issued, if the specified cyclic handler has already been queued into the timer queue, only initialization of the cycle counter is performed.

- *cycact* = `(TCY_ON|TCY_INI)`
  Changes the activity state of the specified cyclic handler to the ON state, queues that handler into the timer queue, then initializes the cycle counter.
  When the activation time is reached, the specified cyclic handler is activated.

- *cycact* = `TCY_ULNK`
  Changes the activity state of the specified cyclic handler to the OFF state, then removes that handler from the timer queue.
  Even when the activation time is reached, the specified cyclic handler is not activated.

Caution **When this system call is issued, if the activity state of the specified cyclic handler is the OFF state, only the queuing of that handler into the timer queue is performed.**

Return value

`E_OK`	`0`	Normal termination

ref_cyc

Refer Cyclic Handler Status

Task/handler

Overview

Acquires cyclic handler information.

C format

`ER 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 cyclic handler information
I	`HNO` *cycno*;	Specification number of cyclic handler

Structure of cyclic 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 cyclic handler information (extended information, remaining time, etc.) of the cyclic handler specified in *cycno* into the packet specified in *pk_rcyc*.

Cyclic handler information is described in detail below.

`exinf` ... Extended information

`lfttim` ... Time remaining until the cyclic handler is next activated (clock interrupt cycles)

`cycact` ... Current activity state

`TCY_OFF(0)`	:	Activity state is OFF.
`TCY_ON(1)`	:	Activity state is ON.
`TCY_ULNK(4)`	:	The cyclic handler is not in the timer queue.

Caution If the value of `cycact` is `TCY_ULNK`, the value of `lfttim` will be undefined.

Return value

`E_OK`	`0`	Normal termination

10.5.7 System Management System Calls

This section explains a group of system calls (system management system calls) that perform processing that is dependent on the system.

Table 10-9 lists the system management system calls.

Table 10-9. List of System Management System Calls

System call	Function	schdl[Note]
`get_ver`	Acquires RX850 version information.	x
`ref_sys`	Acquires system information.	x

Note schdl indicates whether the scheduler is to be activated.
Whether control is to be passed to another task as a result of scheduler activation, however, depends on the current state.
o: Activates the scheduler.
x: Does not activate the scheduler.

get_ver

Get Version Information

Task/handler

Overview

Acquires RX850 version information.

C format

```
ER  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 RX850 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    :  µITRON 3.0 Ver. 3.02
prver   ... OS product version
            H'03xx    :  RX850 Ver. 3.xx
prno[4] ... Product number/product management information
            Undefined :  Serial number of delivery product (each unit has a unique number)
cpu     ... CPU information
            H'0d33    :  V850 family
var     ... Variation descriptor
            H'8000    :  µITRON level S, for single processor use, virtual storage not supported,
                         MMU not supported, file not supported
```

Return value

`E_OK`	`0`	Normal termination

ref_sys

Refer System Status

Task/handler

Overview

Acquires system information.

C format

```
ER ref_sys(T_RSYS *pk_rsys);
```

Parameters

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 permitted.
TTS_DDSP(1)	:	Task processing is being performed. Dispatch processing is inhibited.
TTS_LOC(3)	:	Task processing is being performed. The acceptance of maskable interrupts and dispatch processing is inhibited.
TTS_INDP(4)	:	Processing of a handler (interrupt handler, cyclic handler) is being performed.

Return value

E_OK	0	Normal termination

[MEMO]

APPENDIX A   PROGRAMMING METHODS

This appendix explains how to write processing programs when the CA850 or the CCV850 is being used.

A.1 OVERVIEW

In the RX850, processing programs are classified according to purpose, as shown below.

These processing programs have their own basic formats according to the general conventions or conventions to be applied when the RX850 is used.

(1) Task

The minimum unit of a processing program which can be executed by the RX850.

(2) Directly activated interrupt handler

A routine dedicated to interrupt handling. When an interrupt occurs, this handler is activated without the intervention of the RX850. This routine is handled independently of tasks. When an interrupt occurs, therefore, the processing of the task currently being executed is canceled even if that task has the highest priority relative to any other tasks in the system. Then, control is passed to the directly activated interrupt handler.

In view of direct activation of this handler, a rapid response that approaches the maximum level of hardware performance can be expected.

(3) Indirectly activated interrupt handler

A routine dedicated to interrupt handling. When an interrupt occurs, this handler is activated upon the completion of the preprocessing by the RX850 (such as saving the contents of the registers or switching the stack). This routine is handled independently of tasks. When an interrupt occurs, therefore, the processing of the task currently being executed is canceled even if that task has the highest priority relative to any other tasks in the system. Then, control is passed to the indirectly activated interrupt handler.

The response time of the indirectly activated handler is longer than the directly activated interrupt handler. Because the RX850 completes the required preprocessing, however, the indirectly activated interrupt handler has the advantage of reducing the amount of internal processing to be performed by the handler.

(4) Cyclic 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 the 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 cyclic handler.

A cyclic handler incurs a smaller overhead before the start of execution, relative to any other cyclic processing programs written by the user.

A.2 KEYWORDS

The character strings listed below are reserved as keywords for the configurater. These strings shall not, therefore, be used for other purposes.

auto	clkhdr	cyc	di	ei
flg	flg1	inthdr	intstk	maxpri
mbx	mpf	no_use	no_wait	pool0
Pool1	RX850	rxsers	sem	ser_def
sit_def	TA_MFIFO	TA_MPRI	TCY_ULNK	TCY_OFF
TCY_ON	tsk	tskgrp	TTS_DMT	TTS_RDY
V310				

A.3 RESERVED WORDS

The character strings listed below are reserved as external symbols for the RX850. These strings shall not, therefore, be used for other purposes.

_CYC*	_ID*	inthdrH	inthdrL	Pool0*
Pool1*	RX850*	Sit*	SysIntEnt	Timer_Handler
_txcb*				

Remark * indicates a character string consisting of one or more characters.

A.4 DESCRIBING TASKS

A.4.1 When CA850 Is Used

When describing a task in C, declare the function using a pragma directive and 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 basic format (in C) when the CA850 is used.

Figure A-1. Task Basic Format (in C) When Using CA850

```
#include   <stdrx850.h>


#pragma    rtos_task    func_task
void
func_task  (INT stacd)
{
           /*  Processing of task  func_task   */
           .................................
           .................................
           .................................

           /*  Termination of task func_task  */
           ext_tsk();
}
```

Caution For details of function declaration using a pragma directive, refer to the CA830, CA850 Series C Compiler Package User's Manual, C.

When describing a task in assembly language, describe it as a function conforming to the function call conventions of the CA850.

As an argument (r6 register), the activation code specified when the `sta_tsk` system call is issued is set.

Figure A-2 shows the task basic format (in assembly language) when the CA850 is used.

Figure A-2. Task Basic Format (in Assembly Language) When Using CA850

```
#include   <stdrx850.h>


           .text
           .align       4
           .globl       _func_task
_func_task :
           /*  Processing of task  func_task  */
           .............................
           .............................
           .............................

           /*  Termination of task  func_task  */
           jr           _ext_tsk
```

Cautions 1. When describing a task in assembly language, specify `.c` as the file extension.

2. When describing a task in assembly language, code the following at the beginning of the file:

`#include <stdrx850.h>`

To convert the file to an object file, therefore, preprocessing (front-end processing) must be performed for the CA850.

A.4.2 When CCV850 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 basic format (in C) when the CCV850 is used.

Figure A-3. Task Basic Format (in C) When Using CCV850

```
#include   <stdrx850.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 the CCV850.

As an argument (r6 register), the activation code specified upon the issue of the `sta_tsk` system call is set.

Figure A-4 shows the task basic format (in assembly language) when the CCV850 is used.

Figure A-4. Task Basic Format (in Assembly Language) When Using CCV850

```
#include   <stdrx850.h>


           .text
           .align       4
           .globl       _func_task
_func_task :
           /*   Processing of task  func_task   */
           ..............................
           ..............................
           ..............................

           /*   Termination of task  func_task   */
           jr           _ext_tsk
```

Cautions 1. When describing a task in assembly language, specify `.850` as the file extension.

2. When describing a task in assembly language, code the following at the beginning of the file:
`#include <stdrx850.h>`
To convert the file to an object file, therefore, preprocessing (front-end processing) must be performed for the CCV850.

A.5 DESCRIBING A DIRECTLY ACTIVATED INTERRUPT HANDLER

A directly activated interrupt handler cannot be described in C. Instead, it must be described in assembly language.

Before the processing of a directly activated interrupt handler, the registers must be saved. After the processing, the registers must be restored. The RX850 provides a macro that describes the processing for saving and restoring the registers. This minimizes the workload imposed on the user when he or she describes a handler in assembly language.

The description of the handler differs depending on how the registers are restored (i.e., whether `reti`, `ret_int`, or `ret_wup` is used). Each restoration method is described below, together with the points to be noted.

A.5.1 When Using `reti` (for CA850)

Describe a directly activated interrupt handler as shown in Figure A-5 when using `reti` to restore the registers.

Figure A-5. Example of Description to Restore from a Directly Activated Interrupt Handler Using `reti` (CA850)

```
#include  "stdrx850.inc"
        .section  "interrupt source name"
        jr      _inthdr


        .text
.align        4
        .globl  _inthdr
_inthdr:
        RTOS_IntEntry


        Handler


        RTOS_IntExit
```

`RTOS_IntEntry` informs the RX850 of the start of handler processing. `RTOS_IntExit` informs the RX850 of the completion of the handler processing, and issues the `reti` instruction. These macros do not include a means of saving/restoring the processing performed by registers. Save and restore registers by using the "handler."

The above description cannot be used to switch the stack. The stack of the task that is interrupted is used.

System calls cannot be issued. In addition, an interrupt handler or timer handler that issues a system call cannot be nested. This description is used only for simple processing such as the manipulation of I/O after which execution is immediately returned to the main routine.

A.5.2 When Using `reti` (for CCV850)

Describe a directly activated interrupt handler as shown in Figure A-6 when using `reti` to restore the registers.

Figure A-6. Example of Description to Restore from a Directly Activated Interrupt Handler by Using `reti` (CCV850)

```
#include  <stdrx850.h>


        .org    0x00000150      -- Interrupt entry address
        jr       _inthdr


        .text
.align   4
        .globl  _inthdr
_inthdr:
        RTOS_IntEntry


        Handler


        RTOS_IntExit
```

`RTOS_IntEntry` informs the RX850 of the start of handler processing. `RTOS_IntExit` informs the RX850 of the completion of the handler processing, and issues the `reti` instruction. These macros do not include a means of saving/restoring the processing of registers. Save and restore registers by using the "handler."

The above description cannot be used to switch the stack. The stack of the task that is interrupt is used.

System calls cannot be issued. In addition, an interrupt handler or timer handler that issues a system call cannot be nested. This description is used only for simple processing such as the manipulation of I/O after which execution is immediately returned to the main routine.

Caution When compiling the source program, "`-D_ _asm_ _`" must be used as a compile option.

A.5.3 When Using `ret_int` or `ret_wup` (for CA850)

Describe the handler as shown in Figure A-7 when returning execution from the handler by using `ret_int` or `ret_wup`.

Figure A-7. Example of Description to Restore from a Directly Activated Interrupt Handler by Using `ret_int` or `ret_wup` (CA850)

```
.set     Reg26, 1           --  In 26-register mode
.set     Reg22, 1           --  In 22-register mode
                            --  Unnecessary in 32-register mode
#include  "stdrx850.inc"

        .section "INTP00"   --  Specify an interrupt source name
        jr       _inthdr

        .text
        .globl   _inthdr

        .align 4
_inthdr:
        RTOS_IntEntry
        RTOS_IntPrologue

        .extern _inthdr_body
        jarl       _inthdr_body, lp

        mov     r10, r6

        RTOS_IntEpilogue

        jr      _ret_wup

#include  <stdrx850.h>

ID inthdr_body (void)
{

            Handler

            return tskid;  /* ID number of task to be woken up */
}
```

`RTOS_IntEntry` informs the RX850 of the start of handler processing. `RTOS_IntPrologue` saves the temporary registers and lp, and switches the stack.

Subsequently, registers other than those above (r20 through r30) are saved, and control is transferred to the handler. In Figure A-7, a C function "`inthdr_body`", which is the handler, is called.

The handler (after `RTOS_IntPrologue`) can issue the system call described in **Section 5.3**. Interrupts can also be enabled.

After the processing of the handler has been completed, the registers saved by the user are restored and execution returns from the handler (this is unnecessary in Figure A-7). If the handler is terminated by using `ret_wup`, the ID number of the task to be woken up by the handler is set in register r6.

In Figure A-7, the ID of the task is returned as a return value when execution returns from `inthdr_body`, and its value is copied into r10. Therefore, the value of r10 is copied into r6. When execution is returned from the handler by using `ret_int`, this operation is not necessary.

Once the above processing has been completed, describe macro `RTOS_IntEpilogue`, and terminate the handler by using "`jr ret_int`" or "`jr ret_wup`". Do not describe any processing between `RTOS_IntEpilogue` and "`jr ret_int` (or `ret_wup`)".

Caution For details of the `.section` pseudo instruction, refer to the CA850 C Compiler Package User's Manual, Assembly Language.

A.5.4 When Using `ret_int` or `ret_wup` (for CCV850)

Describe the handler as shown in Figure A-8 when returning execution from the handler by using `ret_int` or `ret_wup`.

Figure A-8. Example of Description to Restore from a Directly Activated Interrupt Handler by Using `ret_int` or `ret_wup` (CCV850)

```
#include  <stdrx850.h>


        .org   0x00000150  --  Interrupt entry address
        jr      _inthdr


        .text
        .globl  _inthdr


        .align  4
_inthdr:
        RTOS_IntEntry
        RTOS_IntPrologue


        .extern _inthdr_body
        jarl    _inthdr_body, lp


        mov     r10, r6


        RTOS_IntEpilogue


        jr      _ret_wup
```

File in which handler is described.

```
#include  <stdrx850.h>


ID inthdr_body (void)
{


        Handler


        return tskid;  /* ID number of task to be woken up */
}
```

`RTOS_IntEntry` informs the RX850 of the start of handler processing. `RTOS_IntPrologue` saves the temporary registers and lp, and switches the stack.

Subsequently, registers other than those above (r20 through r30) are saved, and control is transferred to the handler. In Figure A-8, a C function "`inthdr_body`", which is the handler, is called.

The handler (after `RTOS_IntPrologue`) can issue the system call described in **Section 5.3**. Interrupts can also be enabled.

After the processing of the handler has been completed, the registers saved by the user are restored and execution returns from the handler (this is unnecessary in Figure A-8). If the handler is terminated by using `ret_wup`, the ID number of the task to be woken up by the handler is set in register r6.

In Figure A-8, the ID of the task is returned as a return value when execution returns from `inthdr_body`, and its value is copied into r10. Therefore, the value of r10 is copied into r6. When execution is returned from the handler by using `ret_int`, this operation is not necessary.

Once the above processing has been completed, describe macro `RTOS_IntEpilogue`, and terminate the handler by using "`jr ret_int`" or "`jr ret_wup`". Do not describe any processing between `RTOS_IntEpilogue` and "`jr ret_int` (or `ret_wup`)".

Cautions 1. A branch instruction must be set in a directly activated interrupt handler for the handler address to which processing is passed by the processor upon the occurrence of an interrupt. In Figure A-8, the `.org` instruction section is equivalent to this. The value following the `.org` instruction is equivalent to the handler address of an interrupt.

2. When describing a directly activated interrupt handler in assembly language, specify `.850` as the file extension.

3. When compiling the source program, "`-D_ _asm_ _`" must be used as a compile option.

A.6 DESCRIBING AN INDIRECTLY ACTIVATED INTERRUPT HANDLER

A.6.1 When CA850 Is Used

When describing an indirectly activated interrupt handler in C, describe it as an INT-type function having no argument.

Figure A-9 shows the basic format of an indirectly activated interrupt handler (in C) when the CA850 is used.

Figure A-9. Basic Format of Indirectly Activated Interrupt Handler (in C) When Using CA850

```
#include   <stdrx850.h>


INT
func_inthdr()
{
            /*   Processing of indirectly activated interrupt handler func_inthdr  */
            ........................................................
            ........................................................
            ........................................................

            /*   Notifies the external interrupt controller of the termination of processing   */
            ........................................................

            /*   Return processing from indirectly activated interrupt handler  func_inthdr  */
            return(0xff);
}
```

Caution An indirectly activated interrupt handler is a subroutine called by interrupt preprocessing of the RX850. Therefore, when an indirectly activated interrupt handler is described, an instruction for branching to the indirectly activated 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 indirectly activated interrupt handler in assembly language, describe it as a function conforming to the function call conventions of the CA850.

Figure A-10 shows the basic format of an indirectly activated interrupt handler (in assembly language) when the CA850 is used.

Figure A-10. Basic Format of Indirectly Activated Interrupt Handler (in Assembly Language) When Using CA850

```
#include   <stdrx850.h>


           .text
           .align      4
           .globl      _func_inthdr
_func_inthdr :
           /*  Processing of indirectly activated interrupt handler  func_inthdr   */
           ................................................
           ................................................
           ................................................

           /*  Notifies the external interrupt controller of the termination of processing   */
           ................................................

           /*  Return processing from indirectly activated interrupt handler  func_inthdr   */
           mov         0xff, r10
           jmp         [lp]
```

Cautions 1. When describing an indirectly activated interrupt handler in assembly language, specify `.c` as the file extension.

2. When describing an indirectly activated interrupt handler in assembly language, code the following at the beginning of the file:

`#include <stdrx850.h>`

To convert the file to an object file, therefore, preprocessing (front-end processing) must be performed for the CA850.

3. An indirectly activated interrupt handler is a subroutine called by interrupt preprocessing of the RX850. Therefore, when an indirectly activated interrupt handler is described, an instruction for branching to the indirectly activated interrupt handler need not be set for the handler address to which the processor passes control upon the occurrence of an interrupt.

A.6.2 When CCV850 Is Used

When describing an indirectly activated interrupt handler in C, describe it as an `INT`-type function having no argument.

Figure A-11 shows the basic format of an indirectly activated interrupt handler (in C) when the CCV850 is used.

Figure A-11. Basic Format of Indirectly Activated Interrupt Handler (in C) When Using CCV850

```
#include   <stdrx850.h>


INT
func_inthdr()
{
            /*  Processing of indirectly activated interrupt handler func_inthdr  */
            ..................................................
            ..................................................
            ..................................................

            /*  Notifies the external interrupt controller of the termination of processing  */
            ..................................................

            /*  Return processing from indirectly activated interrupt handler func_inthdr  */
            return(0xff);
}
```

Caution An indirectly activated interrupt handler is a subroutine called by interrupt preprocessing of the RX850. Therefore, when an indirectly activated interrupt handler is described, an instruction for branching to the indirectly activated 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 indirectly activated interrupt handler in assembly language, describe it as a function conforming to the function call conventions of the CCV850.

Figure A-12 shows the basic format of an indirectly activated interrupt handler (in assembly language) when the CCV850 is used.

Figure A-12. Basic Format of Indirectly Activated Interrupt Handler (in Assembly Language) When Using CCV850

```
#include   <stdrx850.h>


           .text
           .align      4
           .globl      _func_inthdr
_func_inthdr :
           /*  Processing of indirectly activated interrupt handler func_inthdr  */
           ...............................................
           ...............................................
           ...............................................

           /*  Notifies the external interrupt controller of the termination of processing  */
           ...............................................

           /*  Return processing from indirectly activated interrupt handler func_inthdr  */
           mov         0xff, r10
           jmp         [lp]
```

Cautions 1. When describing an indirectly activated interrupt handler in assembly language, specify `.850` as the file extension.

2. When describing an indirectly activated interrupt handler in assembly language, code the following at the beginning of the file:

`#include <stdrx850.h>`

To convert the file to an object file, therefore, preprocessing (front-end processing) must be performed for the CCV850.

3. An indirectly activated interrupt handler is a subroutine called by interrupt preprocessing of the RX850. Therefore, when an indirectly activated interrupt handler is described, an instruction for branching to the indirectly activated interrupt handler need not be set for the handler address to which the processor passes control upon the occurrence of an interrupt.

A.7 DESCRIBING CYCLIC HANDLER

A.7.1 When CA850 Is Used

When describing a cyclic handler in C, describe it as a `void`-type function having no argument.
Figure A-13 shows the basic format of a cyclic handler (in C) when the CA850 is used.

Figure A-13. Basic Format of Cyclic Handler (in C) When Using CA850

```
#include   <stdrx850.h>


void
func_cychdr()
{
            /*  Processing of cyclic handler  func_cychdr   */
            ..............................................
            ..............................................
            ..............................................

            /*  Return processing from cyclic handler  func_cychdr   */
            return;
}
```

Caution A cyclic handler is a subroutine called by the time management interrupt handler (clock handler) of the RX850.

When describing a cyclic handler in assembly language, describe it as a function conforming to the function call conventions of the CA850.

Figure A-14 shows the basic format of a cyclic handler (in assembly language) when the CA850 is used.

Figure A-14. Basic Format of Cyclic Handler (in Assembly Language) When Using CA850

```
#include   <stdrx850.h>


           .text
           .align       4
           .globl       _func_cychdr
_func_cychdr :
           /*   Processing of cyclic handler  func_cychdr   */
           .........................................
           .........................................
           .........................................

           /*   Return processing from cyclic handler func_cychdr   */
           jmp          [lp]
```

Cautions 1. When describing a cyclic handler in assembly language, specify `.c` as the file extension.

2. When describing a cyclic handler in assembly language, code the following at the beginning of the file:

`#include <stdrx850.h>`

To convert the file to an object file, therefore, preprocessing (front-end processing) must be performed for the CA850.

3. A cyclic handler is a subroutine called by the time management interrupt handler (clock handler) of the RX850.

A.7.2 When CCV850 Is Used

When describing a cyclic handler in C, describe it as a `void`-type function having no argument.

Figure A-15 shows the basic format of a cyclic handler (in C) when the CCV850 is used.

Figure A-15. Basic Format of Cyclic Handler (in C) When Using CCV850

```
#include   <stdrx850.h>


void
func_cychdr()
{
          /*  Processing of cyclic handler  func_cychdr  */
          ...............................................
          ...............................................
          ...............................................

          /*  Return processing from cyclic handler  func_cychdr  */
          return;
}
```

Caution A cyclic handler is a subroutine called by the time management interrupt handler (clock handler) of the RX850.

When describing a cyclic handler in assembly language, describe it as a function conforming to the function call conventions of the CCV850.

Figure A-16 shows the basic format of a cyclic handler (in assembly language) when the CCV850 is used.

Figure A-16. Basic Format of Cyclic Handler (in Assembly Language) When Using CCV850

```
#include   <stdrx850.h>


           .text
           .align       4
           .globl       _func_cychdr
_func_cychdr :
           /*  Processing of cyclic handler  func_cychdr   */
           ........................................
           ........................................
           ........................................

           /*  Return processing from cyclic handler  func_cychdr   */
           jmp          [lp]
```

Cautions 1. When describing a cyclic handler in assembly language, specify `.850` as the file extension.

2. When describing a cyclic handler in assembly language, code the following at the beginning of the file:

`#include <stdrx850.h>`

To convert the file to an object file, therefore, preprocessing (front-end processing) must be performed for the CCV850.

3. A cyclic handler is a subroutine called by the time management interrupt handler (clock handler) of the RX850.

★ ## A.8 CONSTRAINTS AND NOTES

A.8.1 When Using V850E

The RX850 can be used with the V850E (V850E/MS1, V850E/MA1, V850E/IA1) but its functions are limited. The V850E architecture has an original instruction, CALLT, that is not provided to any other members of the V850 family. This instruction uses two registers: CTPSW and CTPC. The RX850, however, does not save or restore these registers by using a scheduler. If this instruction is used in the program, the operation cannot be guaranteed. Therefore, use of the CALLT instruction is limited.

Do not use the CALLT instruction when describing a task handler in assembly language. When compiling a file in which a task handler is described in C, specify an option that specifies "the CALLT instruction is not used". For details on the compiler option, refer to the manual for each compiler.

A.8.2 Location of `.sit` Section and `.pool0` Section

The RX850 limits the addresses to which the `.sit` section and `.pool0` section can be located. These sections are loaded or stored with a single instruction in r0 (address 0) relative mode, and the range in which they can be located is limited to ±32 Kbytes. Therefore, the range in which these sections can be located is as follows:

- 0xffff8000 to 0xffffffff (However, 0xfffff000 to x0fffffff constitute an internal peripheral I/O area.)
- 0x0 to 0x7fff (However, a few hundred bytes from 0x0 are used as an interrupt handler address area.)

Because the `.sit` section can be embedded in ROM, it should be located at 0x0 to 0x7fff in the internal ROM area of the V850 family. The `.pool0` section must be located in the RAM area, and it is recommended that it be located it in the range of 0xffffe000 to 0xffffefff in the internal RAM area of the V850 family.

If an address to which ROM or RAM is actually allocated (physical address) is specified when linking the applications, however, a link error may occur with some CPUs. For example, with the V850E/MS1, the physical address of the internal RAM area is 0x3ffe0000. If the `.pool0` section is located by using an address in the vicinity of this address, a link error occurs. In this case, link the applications by using the image (mirror image) of the physical address. For details of the image of the physical address, refer to the relevant Hardware Manual.

A.8.3 Range in Which System Calls Can Be Called

The system calls of the RX850 are called by using the jarl instruction. This instruction can access only a 22-bit range because of the architecture, and access is not possible if this range is exceeded by the calling side or called side. Of the 22 bits, the highest bit is a sign bit. Therefore, the positive and negative 21 bits are the limits. Therefore, do not exceed 0x1fffff (2,097,151).

[MEMO]

APPENDIX B INDEX

★

APPENDIX C REVISION HISTORY

A history of revisions up to this edition is shown below. “Applied to:” indicates the chapters to which the revision was applied.

Edition	Contents	Applied to:
2nd edition	Modification of description of target CPU of execution environment	**Chapter 1**
	Modification of description of hardware environment and software environment of development environment	
	Deletion of configurater (GUI section) from the illustration of the system construction procedure	
	Addition of description of constraints and notes	**Appendix A**

[MEMO]

NEC

Facsimile Message

From:

Name

Company

Tel. FAX

Address

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.

Thank you for your kind support.

North America
NEC Electronics Inc.
Corporate Communications Dept.
Fax: +1-800-729-9288
+1-408-588-6130

Europe
NEC Electronics (Europe) GmbH
Technical Documentation Dept.
Fax: +49-211-6503-274

South America
NEC do Brasil S.A.
Fax: +55-11-6462-6829

Hong Kong, Philippines, Oceania
NEC Electronics Hong Kong Ltd.
Fax: +852-2886-9022/9044

Korea
NEC Electronics Hong Kong Ltd.
Seoul Branch
Fax: +82-2-528-4411

Taiwan
NEC Electronics Taiwan Ltd.
Fax: +886-2-2719-5951

Asian Nations except Philippines
NEC Electronics Singapore Pte. Ltd.
Fax: +65-250-3583

Japan
NEC Semiconductor Technical Hotline
Fax: +81- 44-435-9608

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