NEC

User's Manual

RX850 Pro

Real-Time Operating System

Basics

Target Devices
V850 Series™

Target Real-Time OS
RX850 Pro Ver. 3.15

Document No. U13773EJ3V0UM00 (3rd edition)
Date Published April 2003 N CP(K)

[MEMO]

V850 Series, V851, V852, V853, V854, V850/SA1, V850/SB1, V850/SB2, V850/SC1, V850/SC2, V850/SC3, V850/SV1, V850/SF1, V850E/MS1, V850E/MS2, V850E/MA1, V850E/MA2, V850E/IA1, V850E/IA2, V850ES/SA2, V850ES/SA3, V850ES/KF1, V850ES/KG1, and V850ES/KJ1 are trademarks of NEC Electronics Corporation.

MS-DOS, Windows, and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company, Ltd.

PC/AT is a trademark of International Business Machines Corporation.

Green Hills Software and MULTI are trademarks of United States Green Hills Software, Inc.

SPARCstation is a trademark of SPARC International, Inc.

Solaris is a trademark of Sun Microsystems, Inc.

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

ITRON is an abbreviation for Industrial TRON.

μITRON is an abbreviation for "Micro Industrial TRON".

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

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

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

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

  "Specific": Aircraft, aerospace equipment, submersible repeaters, nuclear reactor control systems, life support systems and medical equipment for life support, etc.

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

(Note)

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

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

M8E 02. 11-1

Regional Information

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

J02.11

Major Revisions in This Edition

Page	Description
Throughout	Modification of V850 Family to V850 Series
p.20	Modification of description in **1.7 Execution Environment**
p.21	Modification of description in **1.8 Development Environment**
p.22	Addition of /librxpm.a to nucleus library in **1.9 System Construction Procedure**
p.68	Modification of description of **Caution 2** in **6.3.4 Returning a memory block**
p.80	Modification of description in **7.6.5 Interrupts in cyclically activated handler**
p.193	Modification of description of **Caution** for rel_blk in **11.8.5 Memory pool management system calls**
pp.199, 200	Modification of description of **Structure of system clock SYSTIME** for set_tim and get_tim in **11.8.6 Time management system calls**
p.238	Addition of **APPENDIX B Q & A**

The mark ⋆ shows major revised points.

INTRODUCTION

Target Readers

This manual is intended for users engaged in the design or development of application systems of the V850 Series.

Purpose

This manual is intended to give users an understanding of the functions of the RX850 Pro described in the **Organization** below.

Organization

This manual is roughly organized into the following sections.

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

How to Use This Manual

It is assumed that the reader of this manual has general knowledge in the fields of electrical engineering, logic circuits, microcontrollers, C language, and assembly language.

To learn about the hardware functions or command functions of the V850 Series:
→ Refer to the user's manual for the relevant product.

Conventions

Note:	Footnote for item marked with **Note** in the text
Caution:	Information requiring particular attention
Remark:	Supplementary information
Numerical representation:	Binary ··· XXXX or B'XXXX
	Decimal ··· XXXX
	Hexadecimal ··· 0xXXXX or H'XXXX
Prefix indicating power of 2 (address space, memory capacity):	
	K (kilo) $2^{10} = 1024$
	M (mega) $2^{20} = 1024^2$

Related Documents The following documents may be referenced together with this manual.
The related documents indicated in this publication may include preliminary versions. However, preliminary versions are not marked as such.

Documents related to development tools (user's manuals)

Document Name		Document No.
IE-703002-MC (In-circuit emulator for V851™, V852™, V853™, V854™, V850/SA1™, V850/SB1™, V850/SB2™, V850/SV1™, V850/SF1™, V850/SC1™, V850/SC2™, and V850/SC3™)		U11595E
IE-703003-MC-EM1 (In-circuit emulator option board for V853)		U11596E
IE-703008-MC-EM1 (In-circuit emulator option board for V854)		U12420E
IE-703017-MC-EM1 (In-circuit emulator option board for V850/SA1)		U12898E
IE-703037-MC-EM1 (In-circuit emulator option board for V850/SB1 and V850/SB2)		U14151E
IE-703040-MC-EM1 (In-circuit emulator option board for V850/SV1)		U14337E
IE-703079-MC-EM1 (In-circuit emulator option board for V850/SF1)		U15447E
IE-703089-MC-EM1 (In-circuit emulator option board for V850/SC1, V850/SC2, and V850/SC3)		U15776E
IE-703102-MC (In-circuit emulator for V850E/MS1™ and V850E/MS2™)		U13875E
IE-703102-MC-EM1 (In-circuit emulator option board for V850E/MS1 and V850E/MS2), IE-703102-MC-EM1-A (In-circuit emulator option board for V850E/MS1)		U13876E
IE-V850E-MC (In-circuit emulator for V850E/IA1™ and V850E/IA2™), IE-V850E-MC-A (In-circuit emulator for V850E1 (NB85E core), V850E/MA1™, and V850E/MA2™)		U14487E
IE-V850E-MC-EM1-A (In-circuit emulator option board for V850E1 (NB85E core))		To be prepared
IE-V850E-MC-EM1-B, IE-V850E-MC-MM2 (In-circuit emulator option board for V850E1 (NB85E core))		U14482E
IE-703107-MC-EM1 (In-circuit emulator option board for V850E/MA1 and V850E/MA2)		U14481E
IE-703116-MC-EM1 (In-circuit emulator option board for V850E/IA1)		U14700E
IE-703114-MC-EM1 (In-circuit emulator option board for V850E/IA2)		To be prepared
CA850 Ver. 2.50 C compiler package	Operation	U16053E
	C Language	U16054E
	PM plus	To be prepared
	Assembly Language	U16042E
ID850 Ver. 2.40 Integrated debugger	Operation Windows™ Based	U15181E
SM850 Ver. 2.40 System simulator	Operation Windows Based	U15182E
SM850 Ver. 2.00 or later System simulator	External Part User Open Interface Specifications	U14873E
RX850 Ver. 3.13 or later Real-time OS	Basics	U13430E
	Installation	U13410E
	Technical	U13431E
RX850 Pro Ver. 3.15 Real-time OS	Basics	This manual
	Installation	U13774E
	Technical	U13772E
RD850 Ver. 3.01 Task debugger		U13737E
RD850 Pro Ver. 3.01 Task debugger		U13916E
AZ850 Ver. 3.10 System performance analyzer		U14410E
PG-FP4 Flash memory programmer		U15260E

CONTENTS

LIST OF FIGURES (1/2)

LIST OF FIGURES (2/2)

LIST OF TABLES

CHAPTER 1 OVERVIEW

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

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

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

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

In order to adequately exploit the functions of the V850 Series of high-performance microcontrollers, developed in consideration of the current market conditions, as well as support the creation of more systematic software, NEC Electronics has now released the RX850 Pro.

The RX850 Pro is a control OS for real-time, multitask processing that it has been developed to increase the application range of high-performance, multi-function microcontrollers and further improve their flexibility.

1.1 Overview

The RX850 Pro is an embedded real-time, multitask control OS that provides a highly efficient real-time, multitasking environment to increase the application range of processor control units.

The RX850 Pro is a high-speed, compact OS capable of being stored in and run from the ROM of a target system.

1.2 Real-Time OS

Control equipment demands systems that can rapidly respond to events occurring both internal and external to the equipment. Conventional systems have utilized simple interrupt handling as a means of satisfying this demand. As control equipment has become more powerful, however, it has proved difficult for systems to satisfy these requirements by means of simple interrupt handling alone.

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

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

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

1.3 Multitask OS

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

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

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

One important purpose of a multitask OS is to improve the throughput of the overall system through the parallel processing of multiple tasks.

1.4 Features

The RX850 Pro has the following features.

(1) Conformity with μITRON3.0 specification

As a representative embedded type control OS architecture, the RX850 Pro performs design which is compatible with μITRON3.0 specifications, and includes all the functions up to level E.

The μITRON3.0 specification applies to an embedded, real-time control OS.

(2) High flexibility

The RX850 Pro supports system-specific system calls (7 types), as well as those defined in the μITRON3.0 specification (67 types). The RX850 Pro thus offers superior application system flexibility.

The RX850 Pro can be used to create a real-time, multitask OS that is compact and that can satisfy the user’s needs because the functions (system calls) to be used by the application system can be selected at system construction.

(3) Realization of real-time processing and multitasking

The RX850 Pro 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 Pro supports a function to disable/resume dispatch processing (task scheduling processing). This function enables users to disable/resume dispatch processing from the processing program level.

(5) ROMization

The RX850 Pro is a real-time, multitask OS that has been designed on the assumption that it will be incorporated into the target system, and has been made as compact as possible to enable it to be loaded into a system's ROM.

(6) Use of original instructions

The RX850 Pro realizes high-speed processing through the high-speed instruction execution of the V850 Series microcontrollers and by using original instructions.

(7) Use of internal ROM/RAM

The RX850 Pro realizes high-speed instruction execution and high-speed data access by using the V850 Series internal ROM and RAM.

(8) Application utility support

The RX850 Pro supports the following two utilities to aid in application system construction.

- Configurator CF850 Pro
- High-level language interface library

(9) C compiler

The RX850 Pro supports the following C compilers for the V850 Series.

- CA850 (NEC Electronics)
- CCV850 (Green Hills Software™, Inc.)

1.5 Configuration

The RX850 Pro consists of four subsystems: the nucleus, system initialization, interface library, and system configurator.

These subsystems are outlined below.

(1) Nucleus

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

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

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

(2) System initialization

System initialization includes the hardware initialization and software initialization necessary for the RX850 Pro to run.

When the system is started, therefore, system initialization is executed first.

Among the system initialization processes, sample source files are supplied for the portion that is dependent on the hardware configuration of the execution environment (boot processing and hardware initialization block) and the portion that makes the software environment conformable (software initialization section).

These sample source files improve transplantability to various target systems and facilitate customization.

(3) Interface library

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

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

Furthermore, an interface library compatible with the CA850 C compiler for the NEC Electronics V850 Series and the C cross V800 compiler CCV850 manufactured by Green Hills Software, Inc. are available with the RX850 Pro.

(4) Configurator CF850 Pro

To organize a system using the RX850 Pro, information files holding various data to be supplied to the RX850 Pro (system information table, branch table, and system information header file) are necessary.

Because these files are basically an enumeration of data in specified formats, they can be described by using editors. In this case, however, description and legibility are poor.

Therefore, the RX850 Pro supplies a utility that converts files created in an original description format that has excellent description and legibility (configuration files) to information files.

This utility, the “configurator CF850 Pro”, reads a configuration file created in an original format as an input file and outputs information files such as the system information table, branch table, and system information header file.

1.6 Applications

The RX850 Pro is suitable for the following devices.

- Systems using motor controllers
  PPCs, printers, FAXes
- Systems requiring low power consumption
  Cellular phones, personal handyphones (PHS), digital still cameras

⋆ 1.7 Execution Environment

The RX850 Pro was developed as an OS for embedded control. It therefore runs on a target system equipped with the following hardware.

(1) Operating CPU

- V850 core
- V850E1 core
- V850E2 core
- V850ES core
- V850ES/Kx1 Series
- V850ES/Kx1+ Series

(2) Peripheral controller

The RX850 Pro extracts the hardware-dependent blocks of the nucleus and supplies them as sample source files to support various execution environments. The support is available by rewriting the sample source files for each target system, therefore, no special peripheral controller is required.

(3) Data access to external RAM

The SPOL0 area that is allocated to the RX850 Pro's management area requires support for 8-bit (1-byte) data access. In other words, in cases where 8-bit access to the external RAM area is not supported, SPOL0 area must not be allocated to that area. If it is allocated there, the data being operated on will be lost and normal operation cannot be guaranteed.

As for access to the stack or memory pool, since this does not require 8-bit access, allocation poses no problems as far as the SPOL1 area is concerned.

However, compile options or other means are needed to suppress 8-bit access in the code that accesses data in any memory blocks or other blocks that are created in the SPOL1 area.

This problem can be resolved allocating the SPOL0 area to the internal RAM area in a V850 Series device.

★ 1.8 Development Environment

The hardware and software environments required for system development are shown below.

1.8.1 Hardware environment

(1) Host machine

- PC that runs Windows
- SPARCstation™

(2) In-circuit emulator

Select an in-circuit emulator that is compatible with the CPU to be used. For details, see the relevant pamphlet or other reference document.

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

Select an I/O board that is compatible with the CPU to be used. For details, see the relevant pamphlet or other reference document.

(4) PC interface board

Select a PC interface board that is compatible with the in-circuit emulator and host machine to be used.

1.8.2 Software environment

(1) OS (host machine in parentheses)

- Windows 98, Me, 2000, Windows NT™ 4.0 (PC-9800 series, IBM PC/AT and compatibles)
- Solaris™ 2.x (SPARCstation)

(2) Cross tools

- CA850 (product of NEC Electronics)
- CCV850 (product of GHS)

(3) Debuggers

- ID850 (product of NEC Electronics)
- SM850 (product of NEC Electronics)
- MULTI™, MULTI2000 (product of GHS)
- PARTNER (product of KMC)

(4) Task debugger

- RD850 Pro (product of NEC Electronics)

Remark Included with RX850 Pro package

(5) System performance analyzer

- AZ850 (product of NEC Electronics)

★ 1.9 System Construction Procedure

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

The RX850 Pro system construction procedure is outlined below.

For details, refer to the **RX850 Pro Installation User's Manual (U13774E)**.

(1) Creating a configuration file

(2) Creating an information definition file

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

Note that these information tables are created using a configurator.

(3) Creating system initialization processing

- Boot processing
- Hardware initialization section
- Software initialization section

(4) Creating processing programs

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

These programs are created using C language or assembly language.

(5) Creating an initialization data save area (only when CA850 is used)

(6) Creating a link directive file (section map file)

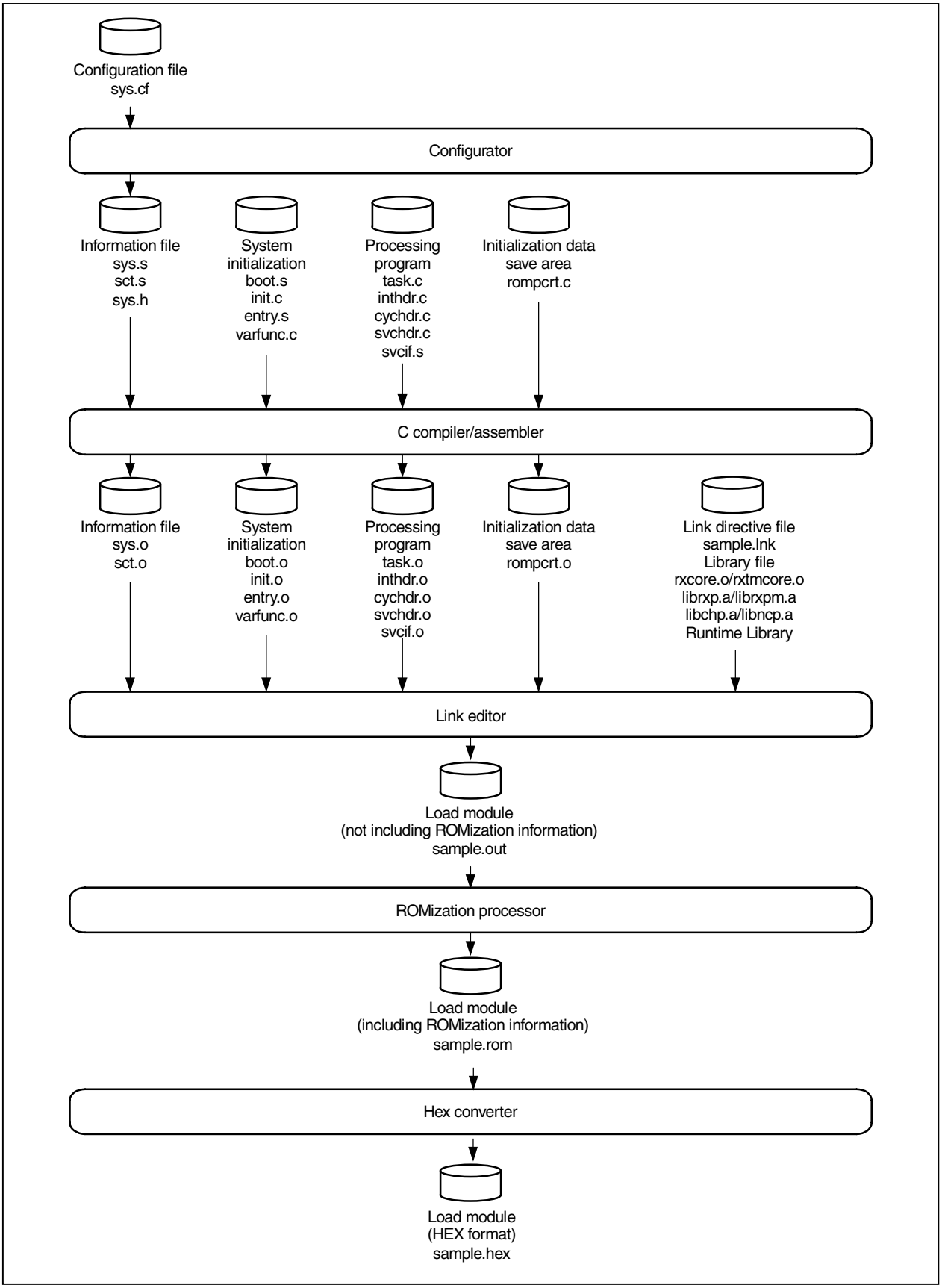
(7) Creating a load module

(8) Incorporating the load module into the system

Remark When using the CCV850 C cross V800 compiler manufactured by Green Hills Software, Inc., the initialization data save area does not need to be created. For details of the creation of the initialization data save area, refer to the **CA850 C Compiler Package Operation User's Manual (U16053E)**.

An example of the system construction procedure when the NEC Electronics V850 Series C compiler CA850 is used is shown in Figure 1-1 and an example of the system construction procedure when the C cross V800 compiler CCV850 manufactured by Green Hills Software, Inc. is used is shown in Figure 1-2.

Figure 1-1. System Construction Procedure (When CA850 Is Used)



The files shown in Figure 1-1 are outlined below. These files are provided as samples.

- **Configuration file**

  sys.cf: Configuration file

- **Information files**

  sys.s: System information table
  sct.s: Branch table
  sys.h: System information header file

- **System initialization**

  boot.s: Boot processing
  init.c: Hardware initialization section (interrupt controller initialization)
  entry.s: Hardware initialization section (interrupt/exception entry)
  varfunc.c: Software initialization section

- **Processing program**

  task.c: Task
  inthdr.c: Interrupt handler
  cychdr.c: Cyclically activated handler
  svchdr.c: Extended SVC handler
  svcif.s: Interface library for an extended SVC handler

- **Initialization data save area**

  rompcrt.s: Initialization data save area

- **Link directive file**

  sample.lnk: Link directive file

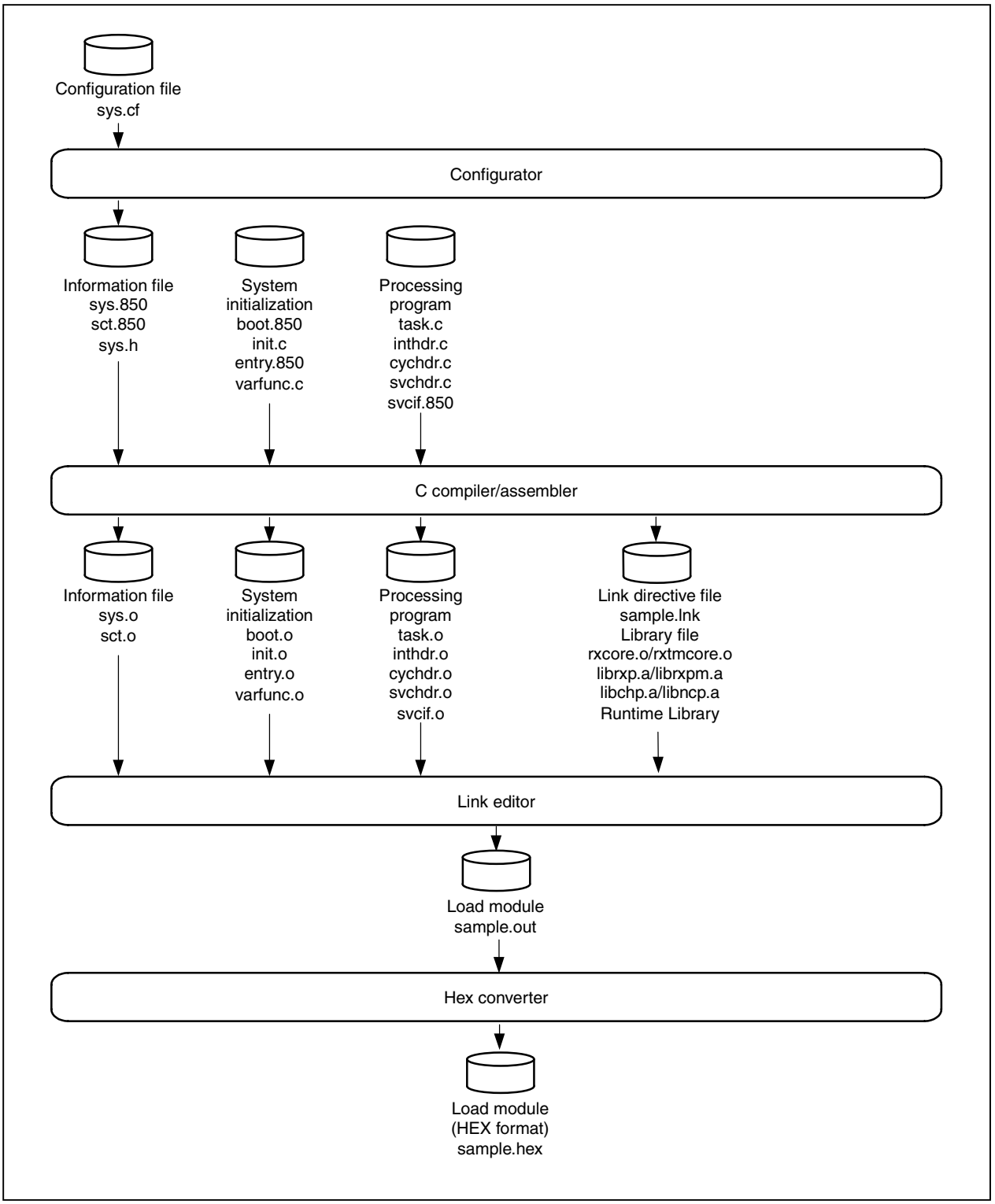
- **Nucleus object**

  rxcore.o/rxtmcore.o: Nucleus common section
  librxp.a/librxpm.a: Nucleus library
  libchp.a/libncp.a: Interface library for system calls

- **Load modules**

  sample.out: Not including ROMization information
  sample.rom: Including ROMization information
  sample.hex: HEX format

Figure 1-2. System Construction Procedure (When CCV850 Is Used)



The files shown in Figure 1-2 are outlined below. These files are provided as samples.

- **Configuration file**

  sys.cf: Configuration file

- **Information files**

  sys.850: System information table
  sct.850: Branch table
  sys.h: System information header file

- **System initialization**

  boot.850: Boot processing
  init.c: Hardware initialization section (interrupt controller initialization)
  entry.850: Hardware initialization section (interrupt/exception entry)
  varfunc.c: Software initialization section

- **Processing program**

  task.c: Task
  inthdr.c: Interrupt handler
  cychdr.c: Cyclically activated handler
  svchdr.c: Extended SVC handler
  svcif.850: Interface library for an extended SVC handler

- **Link directive file**

  sample.lnk: Link directive file

- **Nucleus object**

  rxcore.o/rxtmcore.o: Nucleus common section
  librxp.a/librxpm.a: Nucleus library
  libchp.a/libncp.a: Interface library for system calls

- **Load modules**

  sample.out: Including/not including ROMization information
  sample.hex: HEX format

CHAPTER 2 NUCLEUS

This chapter describes the nucleus, which is the core of the RX850 Pro.

2.1 Overview

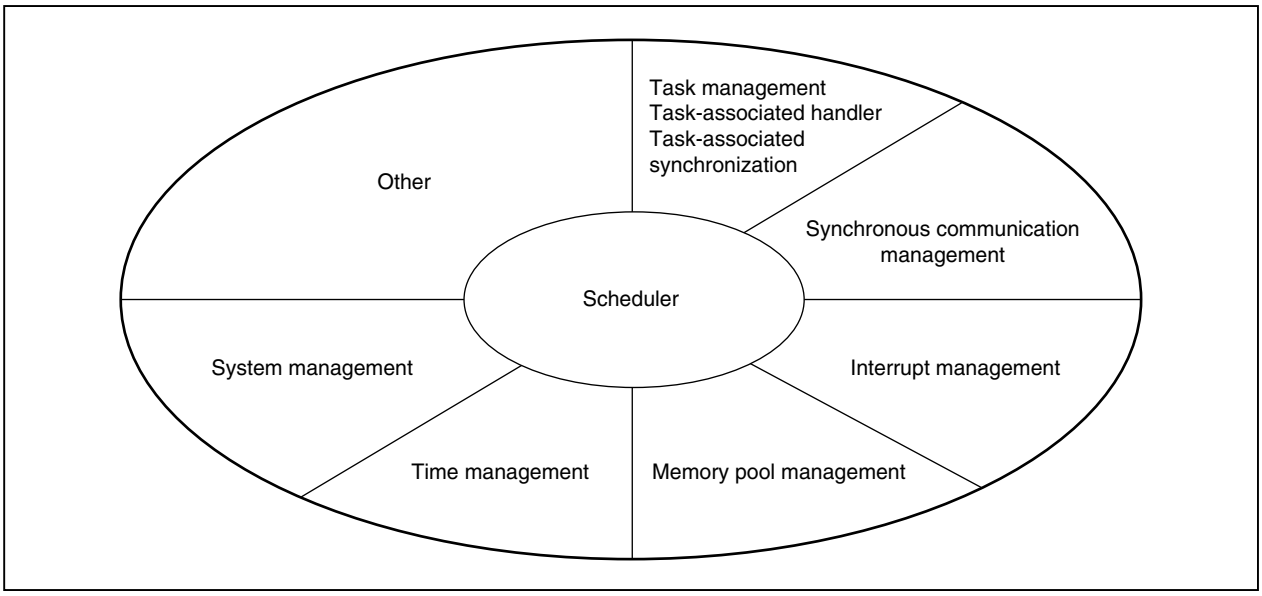
The nucleus forms the heart of the RX850 Pro, a system that supports real-time, multitask control. The nucleus provides the following functions.

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

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

The configuration of the RX850 Pro nucleus is shown below.

Figure 2-1. Nucleus Configuration



2.2 Functions

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

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

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

(1) Task management function

This module manipulates and manages the states of a task, the minimum unit in which processing is performed by the RX850 Pro. For example, the module can create, start, run, stop, terminate, and delete 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
Communication function:	Mailbox

(3) Interrupt management function

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

(4) Memory pool management function

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

- RX850 Pro area
  Management objects
  Memory pool

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

The RX850 Pro also applies dynamic memory pool management. For example, the RX850 Pro provides a function for acquiring and returning a memory area to be used as a work area as required.

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

(5) Time management function

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

(6) Scheduler

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

The RX850 Pro determines the task execution order according to assigned priority levels and by applying the FCFS method. When started, the scheduler determines the priority levels assigned to the tasks, selects an optimum task from those ready to be executed (run or ready state), and optimally assigns tasks a processing time.

Remark In the RX850 Pro, 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 function performed by the RX850 Pro.

3.1 Overview

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

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

3.2 Task States

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

The RX850 Pro classifies task states into the following seven types.

(1) Non-existent state

A task in this state has not been created or has been deleted.

A task in the non-existent state is not managed by the RX850 Pro even if its execution entity is located in memory.

(2) Dormant state

A task in this state has just been created or has already completed its processing.

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

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

- All resources are released.
- The task context is initialized when the processing is resumed.
- A state manipulation system call (ter_tsk, chg_pri, etc.) causes an error.

(3) Ready state

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

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

(4) Run state

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

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

(5) Wait state

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

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

The RX850 Pro further divides tasks in the wait state into the following six groups, according to the conditions which caused the transition to the wait state.

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

(6) Suspend state

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

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

Remark RX850 Pro supports nesting of more than one level of the suspend state (up to 127 times).

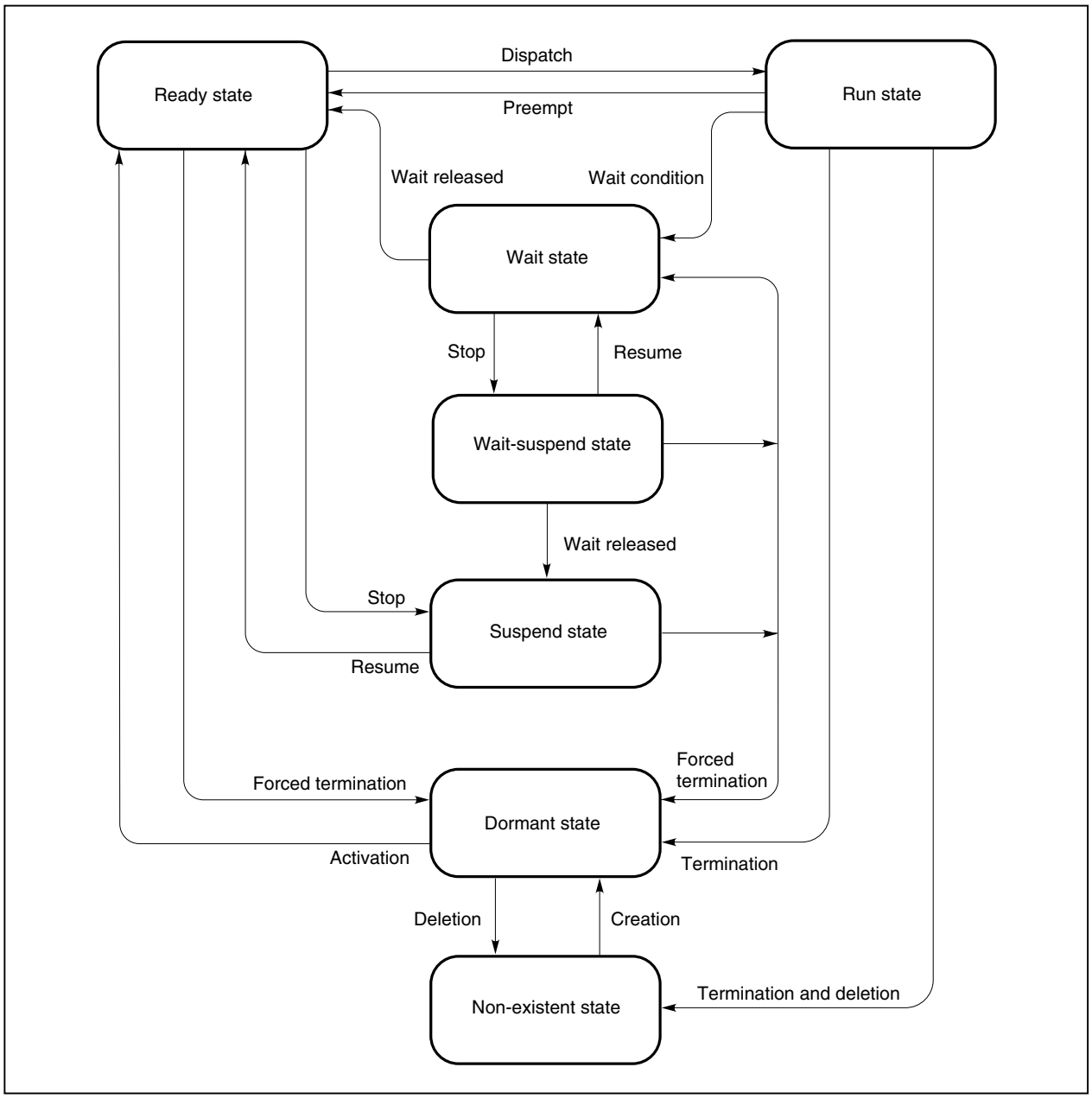
(7) Wait-suspend state

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.

Task state transitions are shown in Figure 3-1.

Figure 3-1. Task State Transition



3.3 Creating Tasks

Two types of interfaces are provided in the RX850 Pro to create tasks: A task is created statically at system initialization (in the nucleus initialization section), or dynamically according to a system call issued from a processing program.

Task in the RX850 Pro consists of three steps: A task management area (management object) is allocated in system memory. Then, the allocated task management area is initialized. Finally, the task state is changed from the non-existent state to the dormant state.

(1) Static registration of a task

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

The RX850 Pro creates a task according to the information defined in the information files (system information table and system information header file) at system initialization, and makes the task manageable.

(2) Dynamic registration of a task

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

The RX850 Pro generates a task according to the information specified with parameters upon the issuance of the cre_tsk system call, and makes the task manageable.

3.4 Activating Tasks

In task activation in the RX850 Pro, a task is switched from the dormant state to the ready state, and scheduled.

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

- sta_tsk system call
  A task specified by the parameters is switched from the dormant state to the ready state.

3.5 Terminating Tasks

In task termination in the RX850 Pro, a task is switched from the ready state, run state, wait state, suspend state, or wait-suspend state to the dormant state and excluded from the schedule by the RX850 Pro.

In the RX850 Pro, a task can be terminated in either of the following two ways.

Normal termination: A task terminates upon completing all processing and when it need not be subsequently scheduled.

Forced termination: When a number of troubles occur during processing and processing must be terminated immediately, this enables termination from another task.

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

- ext_tsk system call
  The task that issued the ext_tsk system call is switched from the run state to the dormant state.

- exd_tsk system call
  The task that issued the exd_tsk system call is switched from the run state to the non-existent state.

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

3.6 Deleting Tasks

In task deletion in the RX850 Pro, a task is switched from the run or dormant state to the non-existent state, and excluded from management by the RX850 Pro.

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

- exd_tsk system call
  The task that issued the exd_tsk system call is switched from the run state to the non-existent state.

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

3.7 Internal Processing of Task

The RX850 Pro utilizes a unique means of scheduling to switch tasks.
Therefore, when describing a task's processing, observe the following points.

(1) Saving/restoring registers

When switching tasks, the RX850 Pro 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 to restore the contents at the end.

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

(2) Stack switching

When switching tasks, the RX850 Pro switches to the special task stack of the selected task. The processing for switching the stack need not be coded at the beginning and end of the task.

(3) Limitations imposed on system calls

Some of the RX850 Pro system calls cannot be issued within a task.
The following system calls can be issued within a task:

- **Task management system calls**

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

- **Task-associated synchronization system calls**

sus_tsk	rsm_tsk	frsm_tsk	slp_tsk	tslp_tsk
wup_tsk	can_wup			

- **Synchronous communication system calls**

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

- **Interrupt management system calls**

def_int	ena_int	dis_int	loc_cpu	unl_cpu
chg_icr	ref_icr			

- **Memory pool management system calls**

cre_mpl	del_mpl	get_blk	pget_blk	tget_blk
rel_blk	ref_mpl	vget_pid		

- **Time management system calls**

set_tim	get_tim	dly_tsk	def_cyc	act_cyc
ref_cyc				

- **System management system calls**

get_ver	ref_sys	def_svc	viss_svc

3.7.1 Acquiring task information

Task information is acquired upon the issuance of the ref_tsk system call.

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

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

3.7.2 Acquiring ID number

The ID number of a task can be acquired by issuing the vget_tid system call.

- vget_tid system call
  Acquires the ID number of the task specified by the parameter.

To manipulate a task with a system call, the ID number of the task is necessary. Whether the ID number is determined univocally by the user or automatically assigned can be specified when a task is created. If automatic assignment of the ID number is specified, however, the user cannot learn the ID number of a task. To do so, a “key ID number” is necessary. The key ID number is univocally specified when a task is created.

By issuing the vget_tid system call with this key ID number as a parameter, the ID number of the task having that key ID number can be acquired.

CHAPTER 4 SYNCHRONOUS COMMUNICATION FUNCTIONS

This chapter describes the synchronous communication functions performed by the RX850 Pro.

4.1 Overview

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

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

In the RX850 Pro, these functions are called “synchronization functions.” The synchronization functions include an exclusive control function and a wait function. The RX850 Pro provides semaphores that act as the exclusive control function and event flags that act as the wait function.

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

In the RX850 Pro, this function is called a “communication function”. The RX850 Pro provides mailboxes that act as the communication function.

4.2 Semaphores

Multitasking requires a function to prevent the resource contention that would occur when concurrently operating multiple tasks attempt to use a limited number of resources such as an A/D converter, coprocessors, files, and programs. To implement this contention preventive function, the RX850 Pro provides non-negative counter-type semaphores.

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

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

Remark In RX850 Pro, those elements required to execute tasks are called resources. In other words, resources comprehensively refer to hardware components such as the A/D converter and coprocessor, as well as software components such as files and programs.

4.2.1 Generating semaphores

The RX850 Pro provides two interfaces for generating semaphores. One enables the static generation of a semaphore during system initialization (in the nucleus initialization section). The other dynamically generates a semaphore by issuing a system call from within a processing program.

To generate a semaphore in the RX850 Pro, an area in system memory is allocated for managing that semaphore (as an object of management by the RX850 Pro), then initialized.

(1) Static registration of a semaphore

To statically register a semaphore, specify it during configuration.

The RX850 Pro generates that semaphore according to the semaphore information defined in the information file (including system information tables and system information header files) during system initialization. The semaphore is subsequently managed by the RX850 Pro.

(2) Dynamic registration of a semaphore

To dynamically register a semaphore, issue the cre_sem system call from within a processing program (task). The RX850 Pro generates that semaphore according to the information specified with parameters when the cre_sem system call is issued. The semaphore is subsequently managed by the RX850 Pro.

4.2.2 Deleting semaphores

A semaphore is deleted by issuing the del_sem system call.

- del_sem system call

  The del_sem system call deletes the semaphore specified by the parameter.

  That semaphore is then no longer managed by the RX850 Pro.

  If a task is queued into the wait queue of the semaphore specified by this system call parameter, that task is removed from the wait queue, after which it leaves the wait state (the resource wait state) and enters the ready state.

  E_DLT is returned to the task released from the wait state as the value returned in response to the system call (wai_sem or twai_sem) that triggered the transition of the task to the wait state.

4.2.3 Returning resources

A resource is returned by issuing the sig_sem system call.

- sig_sem system call

  By issuing the sig_sem system call, the task returns a resource to the semaphore specified by the parameter (the semaphore counter is incremented by 0x1).

  If a task or tasks are queued into the wait queue of the semaphore specified by these system call parameter, the relevant resource is passed to the first task in the wait queue without being returned to the semaphore (thus, the semaphore counter is not incremented).

  That task is then removed from the wait queue, after which it either leaves the wait state (the resource wait state) and enters the ready state, or leaves the wait-suspend state and enters the suspend state.

4.2.4 Acquiring resources

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

- wai_sem system call

  By issuing the wai_sem system call, the task acquires a resource from the semaphore specified by the parameter (the semaphore counter is decremented by 0x1.)

  After issuing this system call, if the task cannot acquire the resource from the specified semaphore (no idle resource exists), the task itself is queued into the wait queue of this semaphore. Thus, the task leaves the run state and enters the wait state (the resource wait state).

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

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

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

- preq_sem system call

  By issuing the preq_sem system call, the task acquires a resource from the semaphore specified by the parameter (the semaphore counter is decremented by 0x1.)

  After this system call is issued, if the task cannot acquire the resource from the specified semaphore (no idle resource exists), E_TMOUT is returned as the return value.

- twai_sem system call

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

  After issuing this system call, if the task cannot acquire the resource from the specified semaphore (no idle resource exists), the task itself is queued into the wait queue of this semaphore. Thus, the task leaves the run state and enters the wait state (the resource wait state).

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

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

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

4.2.5 Acquiring semaphore information

Semaphore information is acquired by issuing the ref_sem system call.

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

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

4.2.6 Acquiring ID number

The ID number of a semaphore can be acquired by issuing the vget_sid system call.

- vget_sid system call
  Acquires the ID number of a semaphore specified by the parameter.

To manipulate a semaphore with a system call, the ID number of the semaphore is necessary. Whether the ID number is determined univocally by the user or automatically assigned can be specified when a task is created. If automatic assignment of the ID number is specified, however, the user cannot learn the ID number of a semaphore. To do so, a “key ID number” is necessary. The key ID number is univocally specified when a semaphore is created.

By issuing the vget_sid system call with this key ID number as a parameter, the ID number of the semaphore having that key ID number can be acquired.

4.2.7 Exclusive control using semaphores

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

Conditions

- Task priority
  Task A > Task B
- State of tasks
  Task A: Run state
  Task B: Ready state
- Semaphore attributes
  Number of resources initially assigned to the semaphore: 0x1
  Maximum number of resources that can be assigned to the semaphore: 0x5
  Task queuing order: FIFO

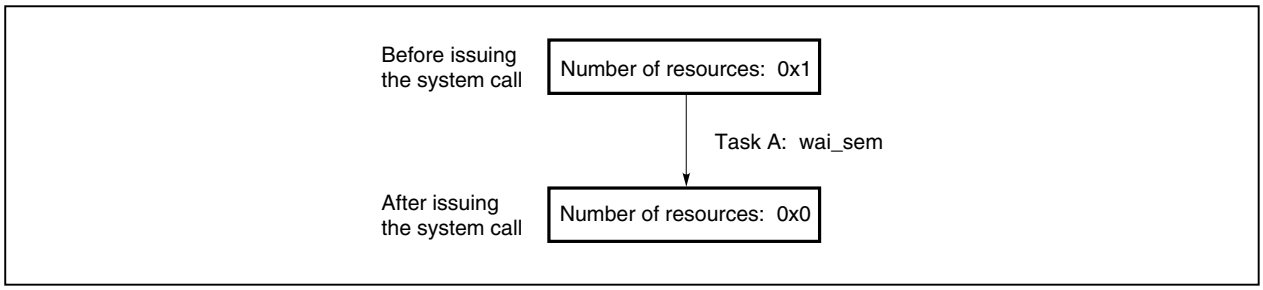
(1) Task A issues the wai_sem system call.

The number of resources assigned to this semaphore and managed by the RX850 Pro is 0x1. Thus, the RX850 Pro decrements the semaphore counter by 0x1.

At this time, task A does not enter the wait state (the resource wait state). Instead, it remains in the run state.

The relevant semaphore counter changes as shown in Figure 4-1.

Figure 4-1. State of Semaphore Counter

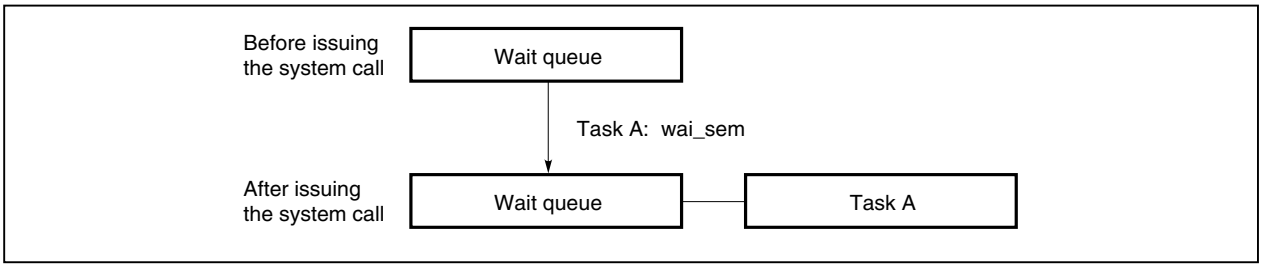


(2) Task A issues the wai_sem system call.

The number of resources assigned to this semaphore and managed by the RX850 Pro is 0x0. Thus, the RX850 Pro changes the state of task A from run to the wait state (resource wait state) and places the task at the end of the wait queue for this semaphore.

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

Figure 4-2. State of Wait Queue (When wai_sem Is Issued)



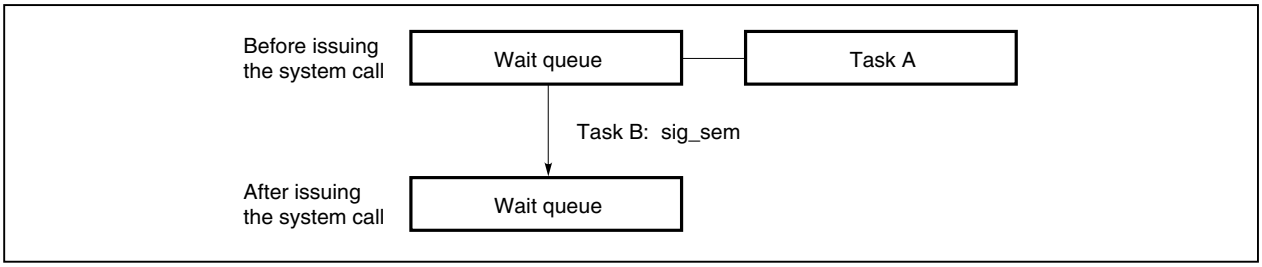
(3) As task A enters the resource wait state, the state of task B changes from ready to run.

(4) Task B issues the sig_sem system call.

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

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

Figure 4-3. State of Wait Queue (When sig_sem Is Issued)

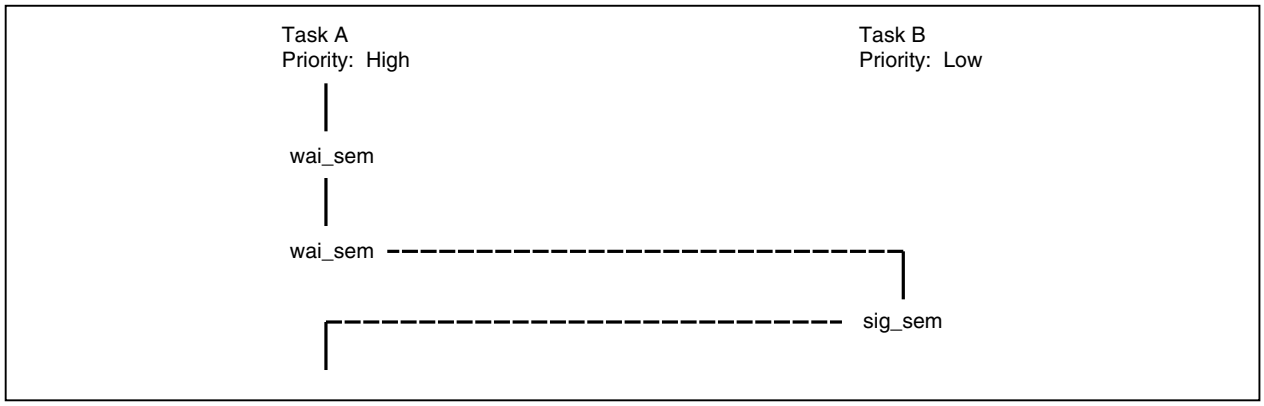


(5) The state of task A having the higher priority changes from ready to run.

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

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

Figure 4-4. Exclusive Control Using Semaphores



4.3 Event Flags

In multitasking, an intertask wait function, in which other tasks wait to resume execution of processing until the results of processing by a given task are output, is necessary. In such a case, it is good to have a function for other tasks to judge whether or not the “processing results output” event has occurred or not, and in the RX850 Pro, an event flag is provided in order to realize this kind of function.

An event flag is a set of data consisting of 1-bit flags that indicate whether a particular event has occurred. 32-bit event flags are used in the RX850 Pro. 32 bits are handled as a set of information with each bit or a combination of bits having a specific meaning.

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

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

4.3.1 Generating event flags

The RX850 Pro provides two interfaces for generating event flags. One is for statically generating an event flag during system initialization (in the nucleus initialization section). The other is for dynamically generating an event flag by issuing a system call from within a processing program.

To generate an event flag in the RX850 Pro, an area in system memory is allocated for managing that event flag (as an object of management by the RX850 Pro), then initialized.

(1) Static registration of an event flag

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

The RX850 Pro generates that event flag according to the event flag information defined in the information file (including system information tables and system information header files) during system initialization. Subsequently, the event flag is managed by the RX850 Pro.

(2) Dynamic registration of an event flag

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

The RX850 Pro generates that event flag according to the information specified by a parameter when the cre_flg system call is issued. Subsequently, the event flag is managed by the RX850 Pro.

4.3.2 Deleting event flags

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

- del_flg system call
  The del_flg system call deletes the event flag specified by the parameter.
  That event flag is then no longer managed by the RX850 Pro.
  If a task is queued into the wait queue of the event flag specified by this system call parameter, that task is removed from the wait queue, after which it leaves the wait state (the event flag wait state) and enters the ready state.
  E_DLT is returned to the task released from the wait state as the return value for the system call (wai_flg or twai_flg) that triggered the transition of the task to the wait state.

4.3.3 Setting a bit pattern

The event flag bit pattern is set by issuing the set_flg system call.

- set_flg system call
  The set_flg system call sets a bit pattern for the event flag specified by the parameter.
  When this system call is issued, if the given condition for a task queued into the wait queue of the specified event flag is satisfied, that task is removed from the wait queue.
  The task then either leaves the wait state (the event flag wait state) and enters the ready state, or leaves the wait-suspend state and enters the suspend state.

4.3.4 Clearing a bit pattern

The event flag bit pattern is cleared by issuing the clr_flg system call.

- clr_flg system call
  The clr_flg system call clears the bit pattern of the event flag specified by the parameter.
  Note that 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.

4.3.5 Checking a bit pattern

The event flag bit pattern is checked by issuing the wai_flg, pol_flg, or twai_flg system call.

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

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

- pol_flg system call
  The pol_flg system call checks whether the bit pattern is set to satisfy the wait condition required for the event flag specified by the parameter.
  If the bit pattern does not satisfy the wait condition required for the event flag specified by this system call parameter, E_TMOUT is returned as the return value.

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

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

Also, the event flag wait conditions and processing when the conditions are established can be specified as follows in the RX850 Pro.

(1) Wait conditions

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

(2) When the condition is satisfied

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

4.3.6 Acquiring event flag information

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

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

  - Extended information
  - Whether tasks are queued
  - Current bit pattern
  - Key ID number

4.3.7 Acquiring ID number

The ID number of an event flag can be acquired by issuing the vget_fid system call.

- vget_fid system call
  Acquires the ID number of the event flag specified by the parameter.

To manipulate an event flag with a system call, the ID number of the event flag is necessary. Whether the ID number is determined univocally by the user or automatically assigned can be specified when an event flag is created. If automatic assignment of the ID number is specified, however, the user cannot learn the ID number of an event flag. To do so, a “key ID number” is necessary. The key ID number is univocally specified when an event flag is created.

By issuing the vget_fid system call with this key ID number as a parameter, the ID number of the event flag having that key ID number can be acquired.

4.3.8 Wait function using event flags

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

Conditions

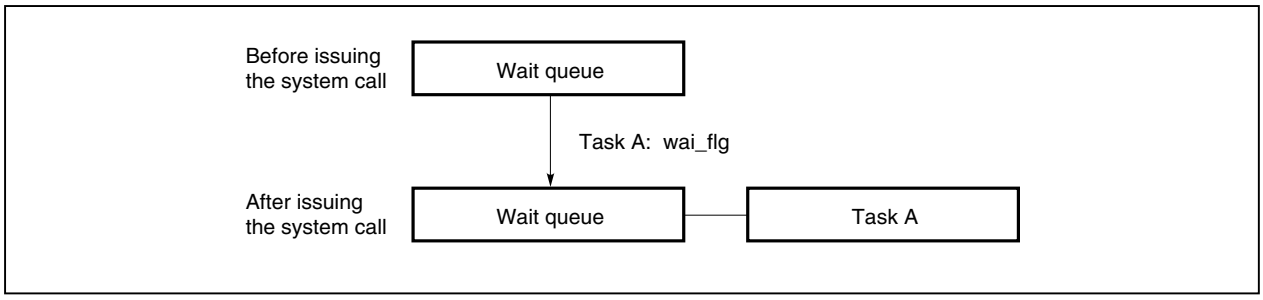
- Task priority
  Task A > Task B
- State of tasks
  Task A: Run state
  Task B: Ready state
- Event flag attributes
  Initial bit pattern: 0x0
  The number of tasks that can be placed in the wait queue: One task

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

The current bit pattern of the relevant event flag managed by the RX850 Pro is 0x0. Thus, the RX850 Pro changes the state of task A from run to wait (the event flag wait state). Task A is then queued at the end of the wait queue for this event flag.

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

Figure 4-5. State of Wait Queue (When wai_flg Is Issued)



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

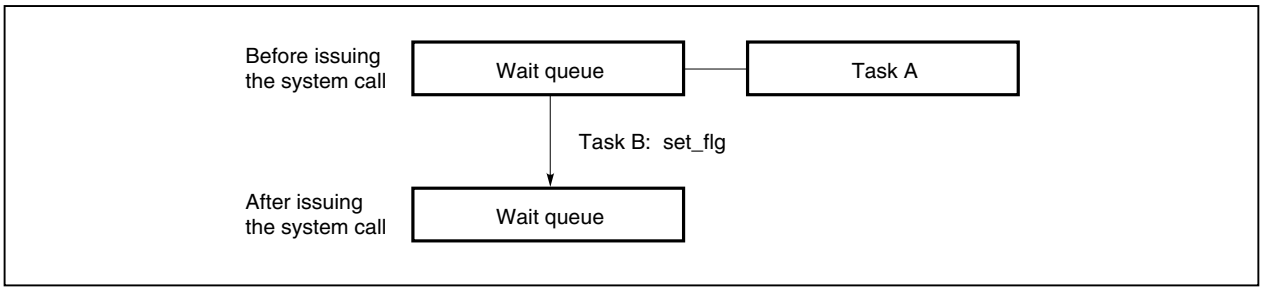
(3) Task B issues the set_flg system call. The bit pattern is set to 0x1.

This bit pattern satisfies the wait condition for task A that has been queued into the wait queue of the relevant event flag. Thus, task A leaves the event flag wait state and enters the ready state.

Since TWF_CLR was specified when task A issued the wai_flag system call, the bit pattern of this event flag is cleared.

The wait queue for this event flag changes as shown in Figure 4-6.

Figure 4-6. State of Wait Queue (When set_flg Is Issued)

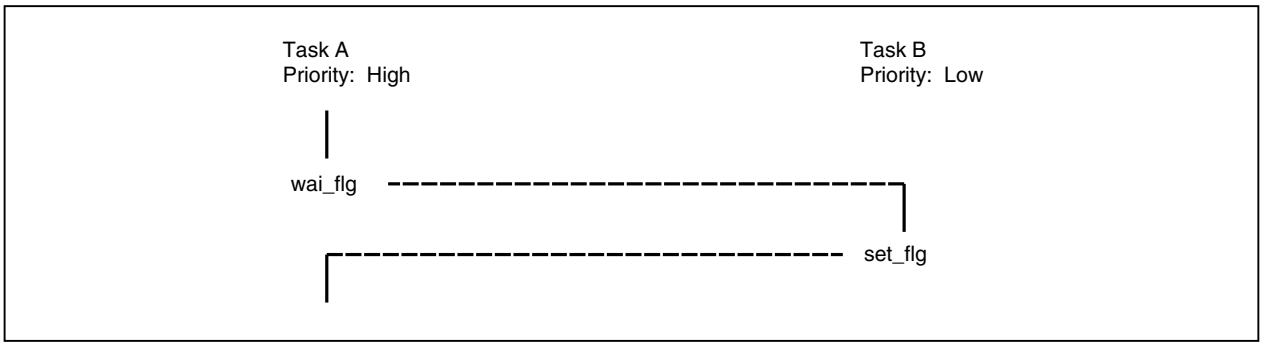


(4) The state of task A having the higher priority changes from ready to run.

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

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

Figure 4-7. Wait and Control by Event Flags



4.4 Mailboxes

Multitasking requires an inter task communication function, so that the tasks can be informed of the results output by other tasks. To implement this function, the RX850 Pro provides mailboxes.

The mailboxes used in the RX850 Pro have two different queues, one dedicated to tasks and the other dedicated to messages. They can be used for both an inter task message communication function and an inter task wait function.

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

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

4.4.1 Generating mailboxes

The RX850 Pro provides two interfaces for generating mailboxes. One is for statically generating a mailbox during system initialization (in the nucleus initialization section). The other is for dynamically generating a mailbox by issuing a system call from within a processing program.

To generate a mailbox in the RX850 Pro, an area in system memory is allocated for managing that mailbox (as an RX850 Pro management object), then initialized.

(1) Static registration of a mailbox

To statically register a mailbox, specify it during configuration.

The RX850 Pro generates the mailbox according to the mailbox information defined in the information file (including system information tables and system information header files) during system initialization. Subsequently, the mailbox is managed by the RX850 Pro.

(2) Dynamic registration of a mailbox

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

The RX850 Pro generates the mailbox according to the information specified by the parameter when the cre_mbx system call is issued. Subsequently, the mailbox is managed by the RX850 Pro.

4.4.2 Deleting mailboxes

A mailbox is deleted by issuing the del_mbx system call.

- del_mbx system call
  The del_mbx system call deletes the mailbox specified by the parameter.
  That mailbox is then no longer managed by the RX850 Pro.
  If a task is queued into the task wait queue of the mailbox specified by this system call parameter, that task is removed from the task wait queue, after which it will leave the wait state (the message wait state) and enter the ready state.
  E_DLT is returned to the task released from the wait state as the return value for the system call (rcv_msg or trcv_msg) that triggered the transition of the task to the wait state.

If a message is queued to the message wait queue of the specified mailbox when this system call is issued, the message is released from the wait queue and is returned to the memory pool from which the message area is acquired. Consequently, if an area other than the memory block acquired from a memory pool is used as a message area, the operation of deleting a mailbox is not guaranteed. Be sure to use a memory block acquired from a memory pool as the message area for the mailbox that may be deleted by this system call.

4.4.3 Transmitting a message

A message is transmitted from the task by issuing the snd_msg system call.

- snd_msg system call
  Upon the issuance of the snd_msg system call, the task transmits a message to the mailbox specified by the parameter.
  If a task or tasks are queued into the task wait queue of the mailbox specified by this system call parameter, the message is delivered to the first task in the task wait queue without being queued into the mailbox.
  The first task is then removed from the wait queue, after which it either leaves the wait state (the message wait state) and enters the ready state, or leaves the wait-suspend state and enters the suspend state.
  If no tasks are queued in the task wait queue of the object mailbox, the message is placed in the message wait queue of the object mailbox.

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

4.4.4 Receiving a message

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

- rcv_msg system call

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

  If the task cannot receive a message from the mailbox specified by this system call parameter (no message exists in the message wait queue of that mailbox), the task that issued this system call is queued at the end of the task wait queue for this mailbox. Thus, the task leaves the run state and enters the wait state (the message wait state).

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

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

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

- prcv_msg system call

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

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

- trcv_msg system call

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

  If the task cannot receive a message from the mailbox specified by this system call parameter (no message exists in the message wait queue for that mailbox), the task that issued this system call is queued at the end of the task wait queue for this mailbox. Thus, the task leaves the run state and enters the wait state (the message wait state).

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

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

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

4.4.5 Messages

In the RX850 Pro, 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 in the RX850 Pro, however, only the start address of a message is delivered to a receiving task, enabling the task to access the message. The contents of the message are not copied to any other area.

(1) Allocating message areas

NEC Electronics recommends that the memory pool managed by the RX850 Pro be allocated for messages. To make a memory pool area available for a message, the task should issue the 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 wait queue when queued. Therefore, save messages after the first four bytes of the message area.

(2) Composition of messages

RX850 Pro does not prescribe the length and composition of messages to be transmitted to mailboxes. The message length, except for the first four bytes, and its composition are defined by the tasks that communicate with each other via mailboxes.

Caution The RX850 Pro prescribes that the first four bytes of each message are used as the block for linkage to the message wait queue when queued. For this reason, when a message is transmitted to the relevant mailbox, the first four bytes of the message must be set to 0x0 before the snd_msg system call is issued.

If the first four bytes of the message are set to a value other than 0x0 when the snd_msg system call is issued, the RX850 Pro determines that this message has already been queued into the message wait queue. Thus, the RX850 Pro does not send the message to the mailbox and returns E_OBJ as the return value.

(3) Priority of messages

The RX850 Pro can specify the priority according to which a message is to be queued. To specify the priority of a message, two bytes are necessary in addition to the four bytes of the link area that is used to queue the message to the message wait queue. Therefore, store the message in an area six bytes after the beginning of the message area. The message priority is specified by a positive integer of 1 to 0x7fff. The lower the value, the higher the priority.

4.4.6 Acquiring mailbox information

Mailbox information is acquired by issuing the ref_mbx system call.

- ref_mbx system call

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

  The mailbox information consists of the following:

  - Extended information
  - Whether tasks are queued
  - Whether messages are queued
  - Key ID number

4.4.7 Acquiring ID number

The ID number of a mailbox can be acquired by issuing the vget_mid system call.

- vget_mid system call
  Acquires the ID number of a mailbox specified by the parameter.

To manipulate a mailbox with a system call, the ID number of the mailbox is necessary. Whether the ID number is determined univocally by the user or automatically assigned can be specified when a mailbox is created. If automatic assignment of the ID number is specified, however, the user cannot learn the ID number of a mailbox. To do so, a "key ID number" is necessary. The key ID number is univocally specified when a mailbox is created.

By issuing the vget_mid system call with this key ID number as a parameter, the ID number of the mailbox having that key ID number can be acquired.

4.4.8 Inter task communication using mailboxes

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

Conditions

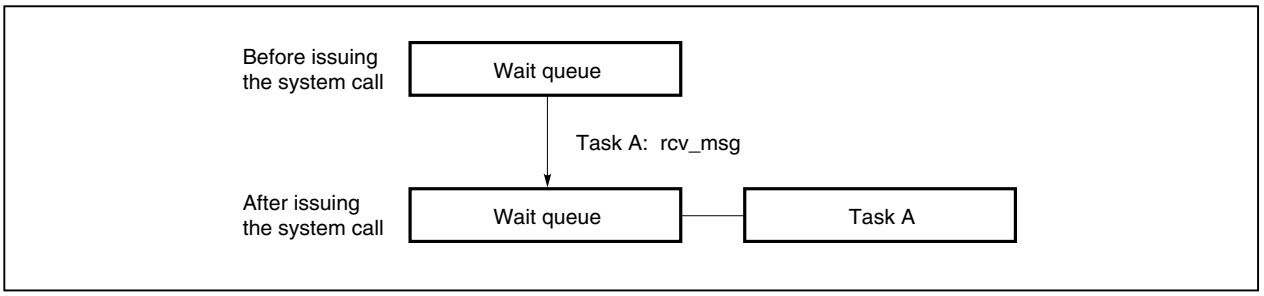
- Task priority
  Task A > Task B
- State of tasks
  Task A: Run state
  Task B: Ready state
- Mailbox attributes
  Task queuing order: FIFO
  Message queuing order: FIFO

(1) Task A issues a rcv_msg system call.

No message is queued in the message wait queue of the relevant mailbox managed by the RX850 Pro. Thus, the RX850 Pro changes the state of task A from run to wait (the message wait state). The task is queued at the end of the task wait queue for this mailbox.

The task wait queue for this mailbox changes as shown in Figure 4-8.

Figure 4-8. State of Task Wait Queue (When rcv_msg Is Issued)



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

(3) Task B issues the get_blk system call.

By means of this system call, a memory pool area is allocated for a message (as a memory block).

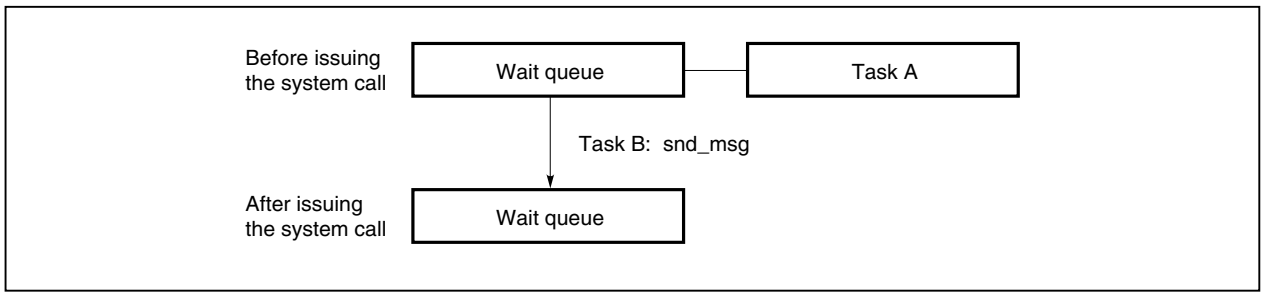
(4) Task B writes a message into this memory block.

(5) Task B issues the snd_msg system call.

This changes the state of task A that has been placed in the task wait for the relevant mailbox from the message wait state to ready state.

The task wait queue for this mailbox changes as shown in Figure 4-9.

Figure 4-9. State of Task Wait Queue (When snd_msg Is Issued)



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

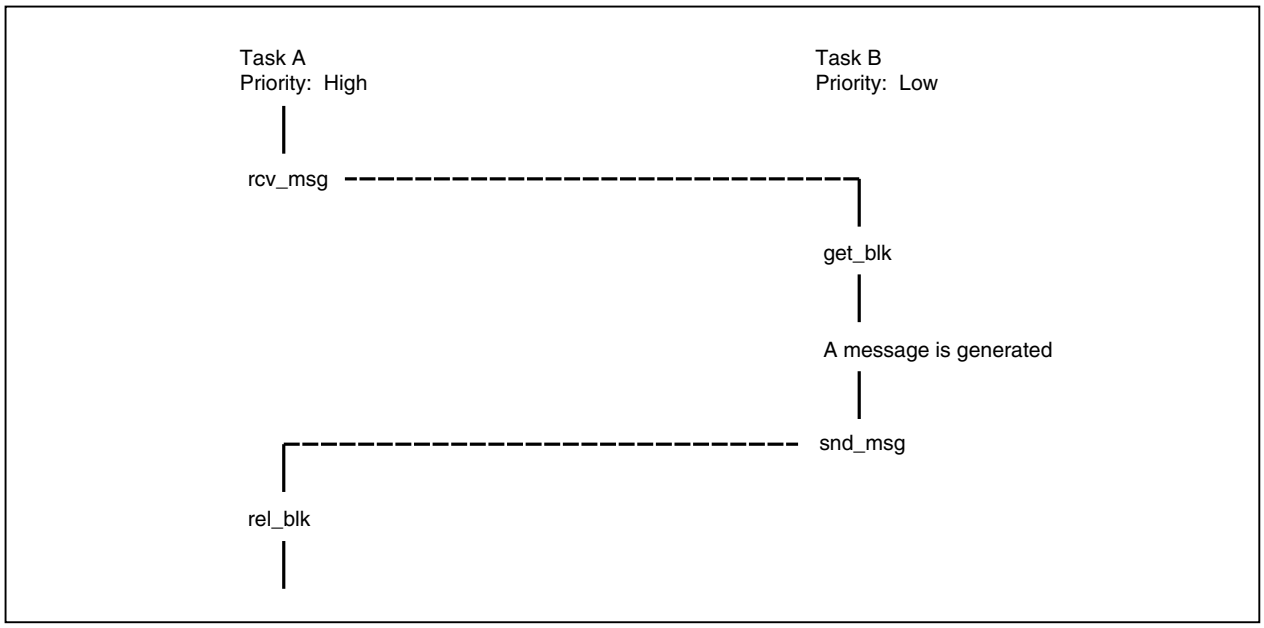
At the same time, task B leaves the run state and enters the ready state.

(7) Task A issues the rel_blk system call.

This releases the memory block allocated for the message in the memory pool.

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

Figure 4-10. Inter-Task Communication Using Mailboxes



CHAPTER 5 INTERRUPT MANAGEMENT FUNCTION

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

5.1 Overview

The RX850 Pro interrupt management function enables the following:

- Registration of an interrupt handler
- Activation of an interrupt handler
- Return from an interrupt handler
- Change or acquisition of the interrupt enable level

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 Pro supports the following two interrupt handler interfaces considering the response from the occurrence of interrupts to the activation of interrupt handler.

- Directly activated interrupt handler
  A routine dedicated to interrupt processing activated without the RX850 Pro.

- Indirectly activated interrupt handler
  A routine dedicated to interrupt processing activated upon the completion of the interrupt preprocessing by the RX850 Pro (such as saving the contents of the registers or switching the stack).

If a system call is issued while the interrupt handler is performing processing, scheduling is performed in a way specific to the RX850 Pro.

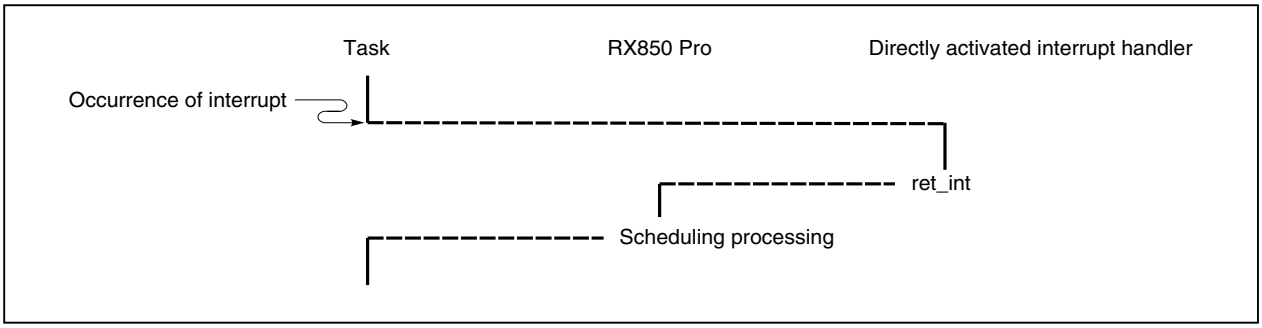
That is, if a system call (chg_pri, sig_sem, etc.) that requires task scheduling is issued during processing by the interrupt handler, the RX850 Pro merely queues the tasks into the wait queue. The actual processing of task scheduling is batched and deferred until a return from the interrupt handler has been made (by issuing the ret_int system call and return instruction).

5.3 Directly Activated Interrupt Handler

A directly activated interrupt handler is a routine dedicated to interrupt processing without using the RX850 Pro upon the occurrence of an interrupt. Accordingly, a high-speed response close to the hardware limitation is expected.

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

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



5.3.1 Registering directly activated interrupt handler

The directly activated interrupt handler is registered by allocating the handler to the handler address to which the processor transfers control if an interrupt occurs, or by setting a branch instruction that branches execution to the directly activated interrupt handler. For details, refer to **A.5 Directly Activated Interrupt Handler**.

5.3.2 Processing in directly activated interrupt handler

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

(1) Saving/restoring the registers

The contents of the work registers when control is transferred to the directly activated interrupt handler are the same as when an interrupt occurred. To use the work registers in the directly activated interrupt handler, therefore, the contents of the work registers must be saved at the start of the handler and restored at the end. To mitigate the workload of the user when describing saving and restoring the work registers in an assembly language, the RX850 Pro provides a macro for the directly activated interrupt handler. This macro helps the user to describe the saving and restoring of the work registers easily.

(2) Stack switching

The stack when control is transferred to the directly activated interrupt handler is the same as the stack when an interrupt occurred. To use the stack for the interrupt handler, therefore, it is necessary to switch the task between that for the handler and that for ordinary purposes at the beginning and end of the indirectly activated interrupt handler. If a macro for the directly activated interrupt handler is used, the stack is switched over between that for the interrupt handler and that for ordinary purposes in the macro.

(3) Limitations imposed on system calls

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

- **Task management system calls**

sta_tsk	chg_pri	rot_rdq	rel_wai	get_tid
ref_tsk	vget_tid			

- **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	vget_sid	set_flg
clr_flg	pol_flg	ret_flg	vget_fid	snd_msg
prcv_msg	ref_mbx	vget_mid		

- **Interrupt management system calls**

def_int	ret_int	ret_wup	ena_int	dis_int
chg_icr	ref_icr			

- **Memory pool management system calls**

pget_blk	rel_blk	ref_mpl	vget_pid

- **Time management system call**

set_tim	get_tim	def_cyc	act_cyc	ref_cyc

- **System management system calls**

get_ver	ref_sys	def_svc	viss_svc

(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
  Performs return from the directly activated interrupt handler.

- ret_wup system call
  Issues a wake-up request to the task specified by the parameters, then returns from the directly activated interrupt handler.

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

Cautions 1. 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 the 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 external interrupt controller before these system calls are issued.

2. The RX850 Pro provides a macro that can be used when a directly activated interrupt handler is described. It is recommended that this macro is also used when execution returns from the handler. In this macro, the necessary registers are restored and the ret_int and ret_wup system calls are issued. To restore the directly activated interrupt handler when this macro is used, therefore, it is not necessary to issue these system calls.
For details, see APPENDIX A PROGRAMMING METHODS.

3. The values of the GP (global pointer) and TP (text pointer) used by the directly activated interrupt handler become undefined. Therefore, the values of the GP and TP should be set again at the start of the directly activated interrupt handler (after macro description). Since the restoring processing is executed by the RX850 Pro, it is not necessary to be described. For details of the description method, refer to A.5 Directly Activated Interrupt Handler.

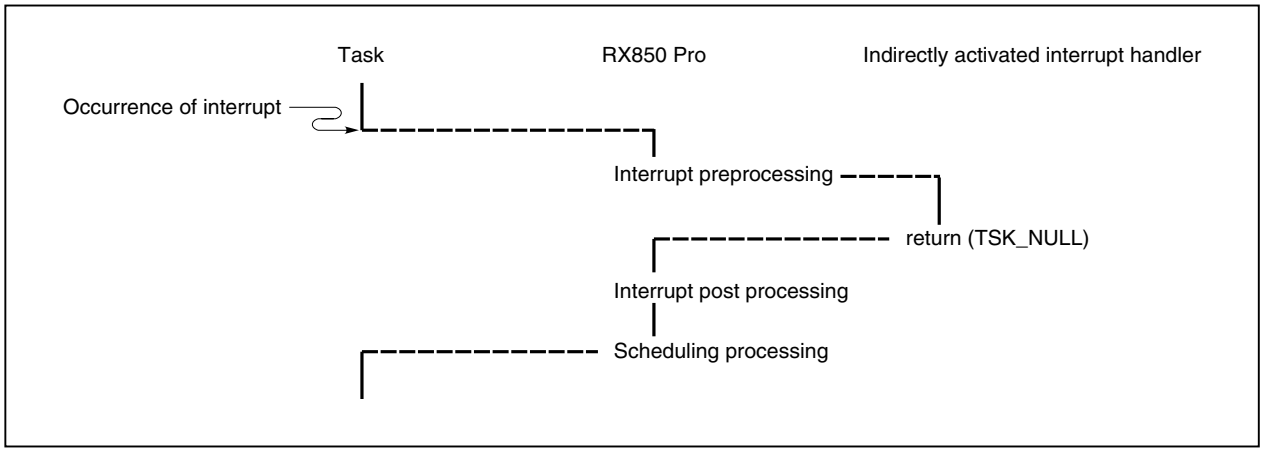
5.4 Indirectly Activated Interrupt Handler

The indirectly activated interrupt handler is an interrupt processing routine that is activated after the interrupt preprocessing of the RX850 Pro (such as saving the registers and switching the stack) has been performed if an interrupt occurs.

This interrupt handler is inferior to the directly activated interrupt handler in terms of response speed. However, because interrupt preprocessing of the RX850 Pro is performed, the processing in the handler is simplified.

Figure 5-2 shows the flow of the operation of the indirectly activated interrupt handler.

Figure 5-2. Operation Flow of Indirectly Activated Interrupt Handler



5.4.1 Registering indirectly activated interrupt handler

The RX850 Pro has two types of interfaces for registering an indirectly activated interrupt handler: “statically register the handler by system initialization (nucleus initialization block)” and “dynamically register the handler by issuing a system call from the processing program”.

Registration of an indirectly activated interrupt handler with the RX850 Pro means allocating an area that manages the indirectly activated interrupt handler (management object) from the system memory and initializing this area.

(1) Static registration

To statically register an indirectly activated interrupt handler, specify so when configuration is executed. The RX850 Pro registers and manages the indirectly activated interrupt handler based on the information defined in the information files (system information table and system information header file) when system initialization is performed.

(2) Dynamic registration

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

The RX850 Pro registers and manages the indirectly activated interrupt handler based on the information specified by the parameter when the def_int system call is issued.

5.4.2 Processing in indirectly activated interrupt handler

Keep in mind the following points when describing the processing of an indirectly activated interrupt handler.

(1) Saving and restoring registers

The RX850 Pro saves and restores the contents of the work registers in compliance with the function calling convention of the C compiler (CA850 or CCV850) when it transfers control to an indirectly activated interrupt handler or when execution returns from the handler. It is therefore not necessary to save the work registers at the beginning of the indirectly activated interrupt handler and to restore the registers at the end.

(2) Switching stack

The RX850 Pro switches the stack when it transfers control to an indirectly activated interrupt handler and when execution returns from the handler. It is therefore not necessary to switch the task between that for the handler and that for ordinary purposes at the beginning and end of the indirectly activated interrupt handler.
If the stack for handler is not defined when configuration is performed, however, the stack is not switched, and the stack for ordinary purposes is used.

(3) Issuing system calls

Here is a list of the system calls that can be issued in the indirectly activated interrupt handler.

- **Task management system calls**

sta_tsk	chg_pri	rot_rdq	rel_wai	get_tid
rer_tsk	vget_tid			

- **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	vget_sid	set_flg
clr_flg	pol_flg	ref_flg	vget_fid	snd_msg
prcv_msg	ref_mbx	vget_mid		

- **Interrupt management system calls**

def_int	ena_int	dis_int	ref_icr	chg_icr

- **Memory pool management system calls**

pget_blk	rel_blk	ref_mpl	vget_pid

- **Time management system calls**

set_tim	get_tim	def_cyc	act_cyc	ref_cyc

- **System management system calls**

get_ver	ref_sys	def_svc	viss_svc

(4) Return processing from indirectly activated interrupt handler

To exit an indirectly activated interrupt handler, issue the return instruction at the end of the handler.

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

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

The RX850 Pro only manipulates the queues if a system call requiring scheduling of a task (such as chg_pri and sig_sem) is issued in an indirectly activated interrupt handler. The actual scheduling is postponed until execution returns from the indirectly activated interrupt handler, and is then performed all at once.

Caution The return instruction does not notify an external interrupt controller of the end of processing (by issuing the EOI command). To exit from an indirectly activated interrupt handler that has been activated by an external interrupt request, therefore, notify the external interrupt controller of the end of the processing before issuing these system calls.

5.5 Disabling/Resuming Maskable Interrupt Acknowledgement

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

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

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

- unl_cpu system call
  The issue of the unl_cpu system call enables the acknowledgement of maskable interrupts, and resuming dispatch processing (task scheduling).
  This system call enables the acknowledgement of maskable interrupts which is disabled by the loc_cpu system call and resumes dispatch processing.

Figure 5-3 shows the flow of control if an interrupt is not masked (normal) and Figure 5-4 shows the flow of control if the loc_cpu system call is issued.

Figure 5-3. Control Flow if Interrupt Mask Processing Is Not Performed (Normal)

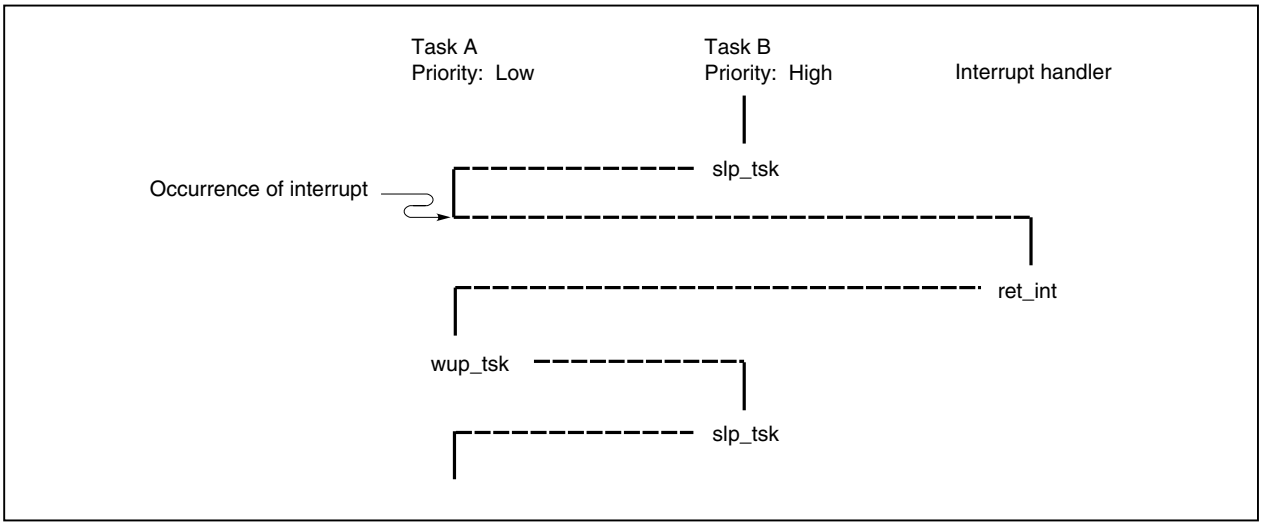
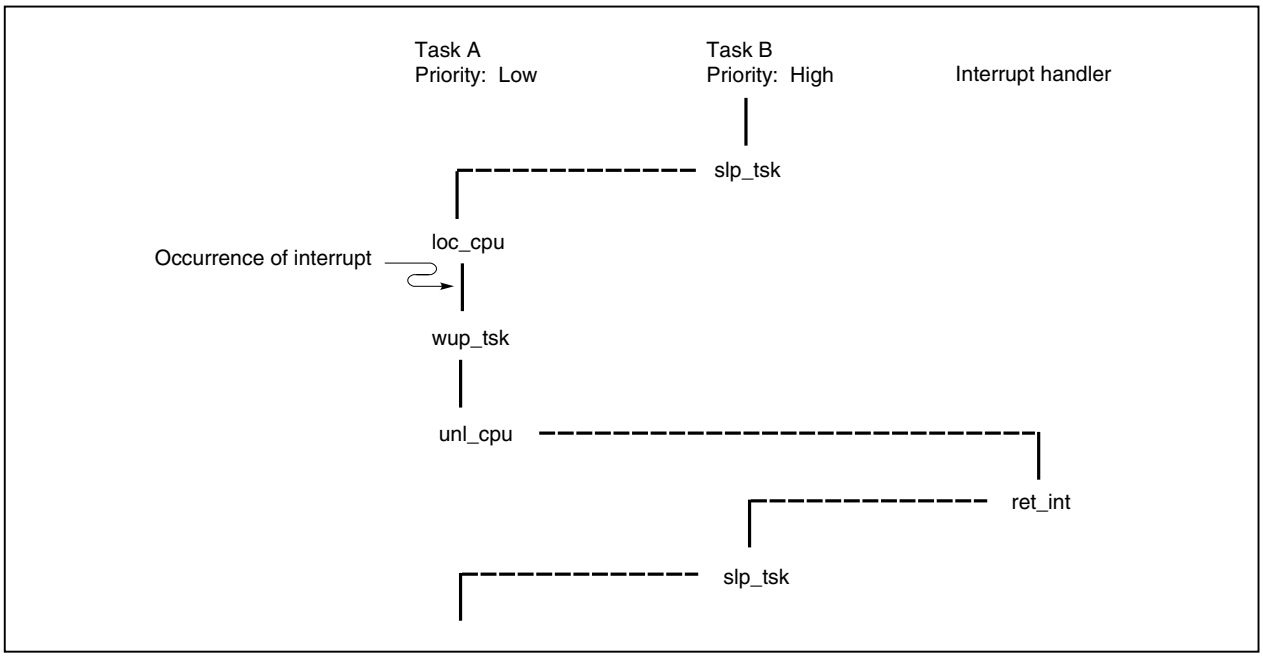


Figure 5-4. Control Flow if loc_cpu System Call Is Issued



5.6 Changing/Acquiring Interrupt Control Register

The interrupt control register is changed or acquired by issuing the chg_icr or ref_icr system call.

- chg_icr system call
  This system call changes the interrupt control register specified by the parameter.

- ref_icr system call
  This system call is available for acquiring the interrupt control register specified by the parameter.

Caution When the RX850 Pro is operated on the V850E core, if the interrupt control register-related system calls chg_icr and ref_icr are issued, the desired interrupt control register may not be operated. In the RX850 Pro, the interrupt control register address is calculated from the interrupt source number. However, in the V850E core, the correct register address cannot be obtained since the alignment of the interrupt source numbers and interrupt control registers differs from other V850 Series products. Therefore, use of the system calls chg_icr and ref_icr is restricted. For manipulating the interrupt control register via an application, directly manipulate the register without using these system calls.

5.7 Non-Maskable Interrupts

A non-maskable interrupt is not subject to management based on interrupt priority and has priority over all other interrupts. It can be acknowledged even if the processor is placed in the interrupt disabled state (setting the ID flag of PSW).

Therefore, even while processing is being executed by the RX850 Pro or an interrupt handler, a non-maskable interrupt can be acknowledged.

If a system call is issued during the processing of an interrupt handler that supports non-maskable interrupts, its operation cannot be assured in the RX850 Pro.

5.8 Clock Interrupts

In the RX850 Pro, time management is performed using clock interrupts, which can be generated periodically by hardware (clock controller, etc.).

If a clock interrupt is issued, RX850 Pro system clock processing is called and the processing related to time, such as the timeout wait of a task or the activation of the cyclically activated handler, is performed.

For details about the time management, see **CHAPTER 7 TIME MANAGEMENT FUNCTION**.

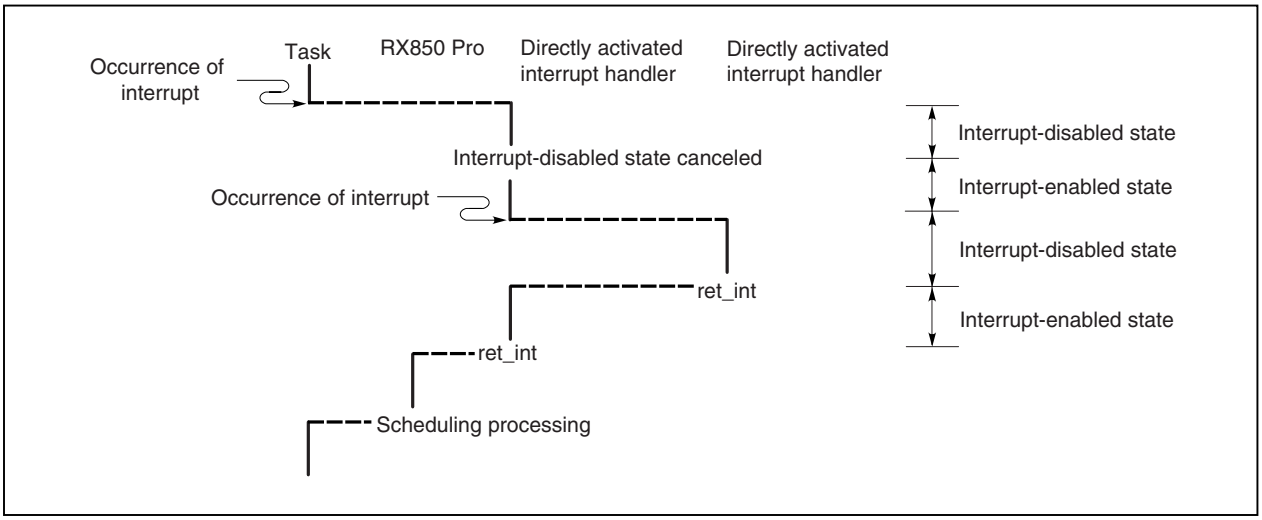
5.9 Multiple Interrupts

The occurrence of another interrupt while processing is being performed by an interrupt handler is called “multiple interrupts”. The RX850 Pro also responds to multiple interrupts.

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

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

Figure 5-5. Processing Flow for Handling Multiple Interrupts



CHAPTER 6 MEMORY POOL MANAGEMENT FUNCTION

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

6.1 Overview

The information table to manage systems, memory areas where the management blocks for implementing functions are allocated, and memory areas to use the memory pool are required for the RX850 Pro.

The items above are allocated in the following four types of memory areas.

- System Memory Pool 0 (Keyword: SPOL0)
- System Memory Pool 1 (Keyword: SPOL1)
- User Memory Pool 0 (Keyword: UPOL0)
- User Memory Pool 1 (Keyword: UPOL1)

The resource management table, task stack, interrupt handler stack, and memory for memory pool are allocated in the above memory areas. The combination of allocatable areas is as follows.

Table 6-1. Memory Information Allocation Combination

Resource Management Table	Task Stack	Interrupt Stack	Memory Pool
SPOL0	SPOL0 or SPOL1	SPOL0 or SPOL1	UPOL0 or UPOL1

The start address and size of each memory area are set using the configuration file. SPOL0 must be created. SPOL1 needs to be created when the task stack and interrupt stack are to be allocated in other than SPOL0. UPOL0 and UPOL1 need to be created if the memory pool management function is to be used. In addition, UPOL1 can be created if UPOL0 has already been created.

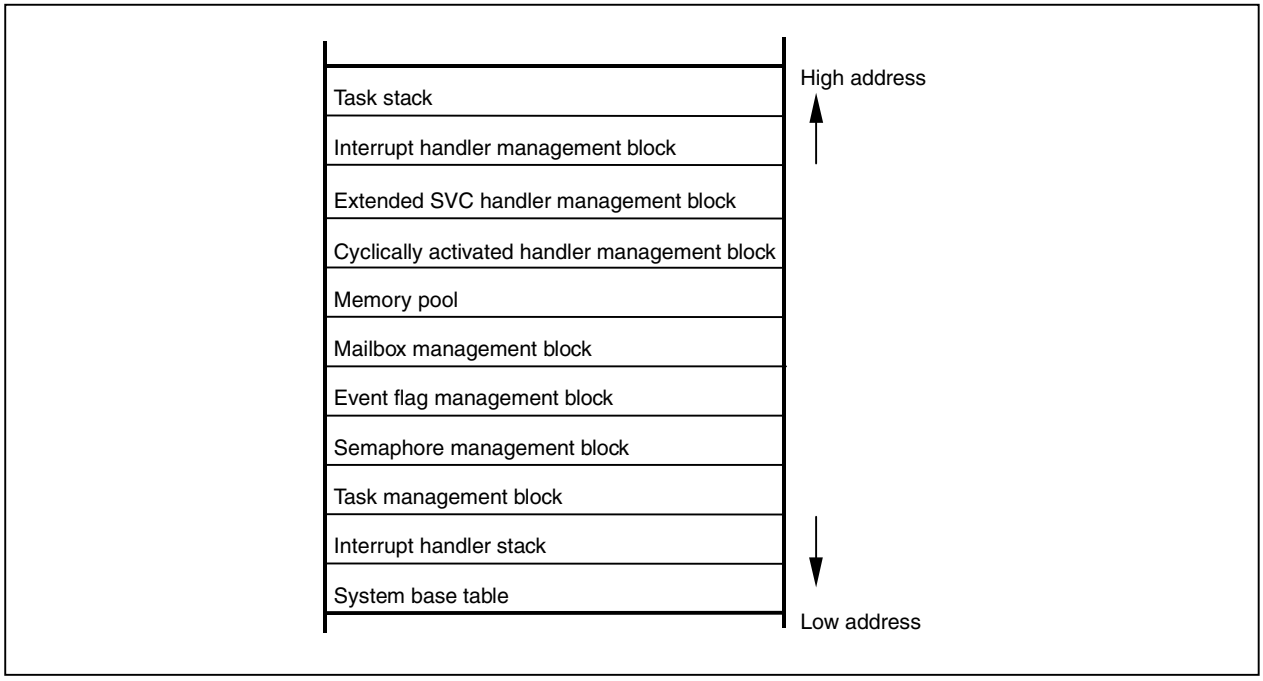
6.2 Management Objects

The objects required for implementing the functions provided by the RX850 Pro are listed below. These management objects are generated and initialized during system initialization, according to the information specified at configuration. These management objects are allocated to SPOL0 (SPOL1 also is available for task stacks and interrupt handler stacks).

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

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

Figure 6-1. Typical Arrangement of Management Objects



6.3 Memory Pool and Memory Blocks

The RX850 Pro executes a dynamic memory pool management function through which memory areas are acquired and released during application. Using this function, the memory area is acquired if required for working, and if it becomes unnecessary, the memory area is released. This function enables efficient use of limited memory area.

The memory area that can be used as a memory pool is UPOL0 or UPOL1. Specify which area of the UPOL0 or UPOL1 is to be used when the memory pool is defined at configuration, or when the memory pool is created by issuing a system call (cre_mpl).

The RX850 Pro provides a variable-length memory pool, but not a fixed-length memory pool.

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

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

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

6.3.1 Generating a memory pool

The RX850 Pro provides two interfaces for generating (registering) a memory pool. One enables static generation during system initialization (in the nucleus initialization section). The other enables dynamic generation by issuing a system call from within a processing program.

To generate a memory pool in the RX850 Pro, certain areas in system memory are allocated to enable management of the memory pool (as an object of RX850 Pro management) and for the memory pool main body, then initialized.

(1) Static registration of a memory pool

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

The RX850 Pro generates the memory pool, based on the information defined in the information file (including system information tables and system information header files) during system initialization. The memory pool is subsequently managed by the RX850 Pro.

(2) Dynamic registration of a memory pool

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

The RX850 Pro generates the memory pool, according to the information specified by the parameters when the cre_mpl system call is issued. The memory pool is subsequently managed by the RX850 Pro.

Remark When a memory pool is created, the RX850 Pro uses the first 8 bytes of the memory pool as a memory pool management area, in addition to the specified size of memory. Therefore, the size of the created memory pool is “specified size + 8 bytes”.

6.3.2 Deleting a memory pool

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

- del_mpl system call
  The del_mpl system call deletes the memory pool specified by the parameter.
  Subsequently, that memory pool is no longer subject to management by the RX850 Pro.
  If a task is queued into the wait queue of the memory pool specified by this system call parameter, that task is removed from the wait queue, leaves the wait state (the memory block wait state) and enters the ready state.
  E_DLT is returned to the task released from the wait state as the return value for the system call (get_blk or tget_blk) that triggered the transition of the task to the wait state.
  If this system call is issued, the RX850 Pro excludes the memory block managed by the specified memory pool from management. If the task has already acquired a memory block from the memory pool before this system call is issued, the operation of the memory block is not guaranteed, and care must be exercised in deleting a memory pool.

6.3.3 Acquiring a memory block

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

Caution In the RX850 Pro, memory clear is not performed when a memory block is acquired. Therefore, the acquired memory block's contents are undefined.

When a memory block is acquired, the RX850 Pro uses eight bytes of the memory pool as a memory management area, in addition to the requested size of memory. The RX850 Pro also aligns the requested size by four bytes. Check the remaining memory block size.

The size of the acquired memory block can be calculated by this expression:

Size of memory block (blk_siz) = align 4 (user requested size) + 8

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

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

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

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

- tget_blk system call

  By issuing a tget_blk system call, the processing program (task) acquires a memory block from the memory pool specified by a parameter.

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

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

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

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

6.3.4 Returning a memory block

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

- rel_blk system call

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

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

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

Cautions 1. The contents of a returned memory block are not cleared by the RX850 Pro. Thus, the contents of a memory block may be undefined when that memory block is returned.

★ **2. The RX850 Pro includes two different specifications for the rel_blk system call.**

(1) When a memory block is returned by a rel_blk system call, if the first four bytes of the memory block are not filled with zeros, the return value E_OBJ is used for termination instead of returning the memory block.

(2) When the rel_blk system call is issued, the memory block is returned even if the first four bytes of the memory block are not filled with zeros (return value = E_OK).

The first specification applies when the memory block is used as a mailbox's message area, and this is the specification that has been used for the rel_blk system call as it has been implemented thus far in the RX850 Pro.

When the memory block is used as a mailbox's message area, the first four bytes serve as the link area for the message's wait queue. In other words, if messages are queued in the mailbox, when the rel_blk system call is issued and the memory block must be returned, in which case it is the message area linked to the queue that is returned. To prevent this, the specification requires the first four bytes that comprise the link area to be filled with zeros, otherwise it will be recognized as the memory block used as the message area and the return value E_OBJ will be used for termination instead of returning the memory block. Under this specification, the first four bytes must be cleared to zero in order to use rel_blk to return the memory block.
These specifications of rel_blk are stored in separate libraries so that one or the other rel_blk specification can be used. Link to the library of the rel_blk specification to be used.

(1) Library containing rel_blk that requires zero-clearing of first four bytes of memory block → librxp.a

(2) Library containing rel_blk that does not require zero-clearing of first four bytes of memory block → librxpm.a

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

6.3.5 Acquiring memory pool information

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

- ref_mpl system call
  Upon the issuance of the ref_mpl system call, the processing program (task) acquires the memory pool information (extended information, queued tasks, etc.) for the memory pool specified by the parameter.
  The memory pool information consists of the following:

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

6.3.6 Acquiring ID number

The ID number of a memory pool can be acquired by issuing the vget_pid system call.

- vget_pid system call
  Acquires the ID number of a memory pool specified by the parameter.

To manipulate a memory pool with a system call, the ID number of the memory pool is necessary. Whether the ID number is determined univocally by the user or automatically assigned can be specified when a memory pool is created. If automatic assignment of the ID number is specified, however, the user cannot learn the ID number of a memory pool. To do so, a “key ID number” is necessary. The key ID number is univocally specified when a memory pool is created.

By issuing the vget_pid system call with this key ID number as the parameter, the ID number of the memory pool having that key ID number can be acquired.

6.3.7 Dynamic management of memory block by memory pool

Here is an example of an operation to dynamically use the memory for tasks by using a memory pool.

Conditions

- Task priority
  Task A > Task B
- State of tasks
  Task A: Run state
  Task B: Ready state
- Memory pool attributes
  Vacant memory block size: 0x20
  Task queuing order: FIFO

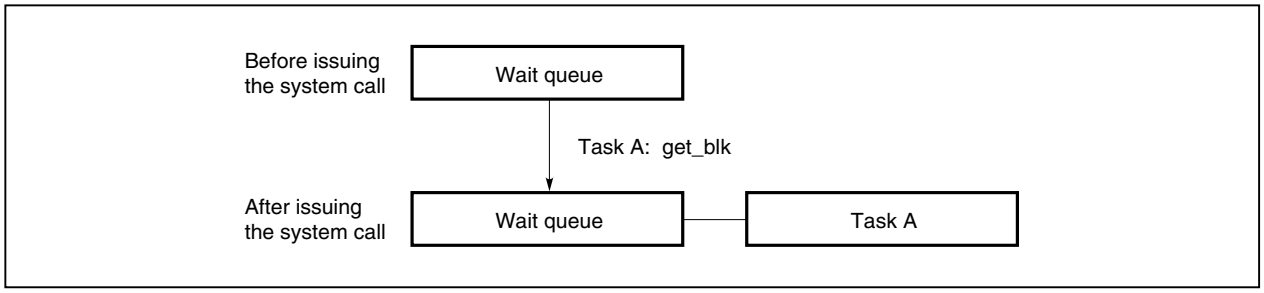
(1) Task A issues the get_blk system call.

The requested memory block size is “0x30”.

At present, the vacant memory block size of the memory pool under management of the RX850 Pro is “0x20”. Therefore, the RX850 Pro changes the state of task A from run to wait (waiting for a memory block), and queues the task to the end of the wait queue of tasks waiting for a memory pool.

At this time, this wait queue is in the state as shown in Figure 6-2.

Figure 6-2. State of Wait Queue (When get_blk Is Issued)



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

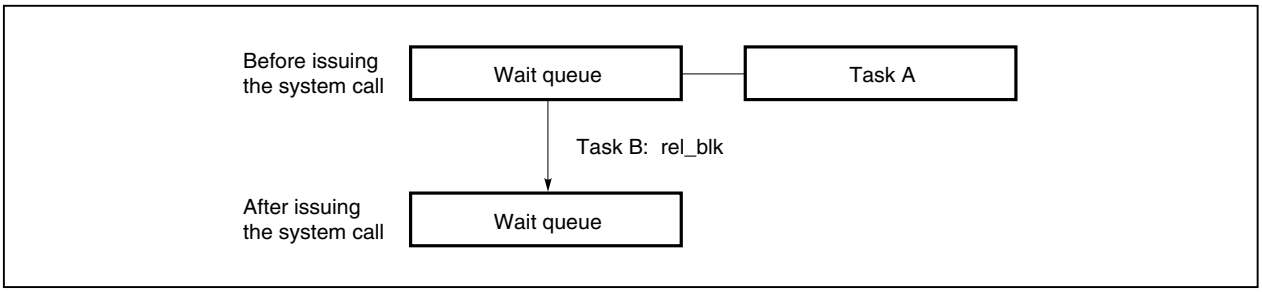
(3) Task B issues the rel_blk system call.

The returned memory block size is “0x16”.

As a result, the requested memory block size of task A queued waiting for a memory pool is satisfied and task A changes its state from wait to ready.

At this time the wait queue of tasks waiting for a memory pool is as shown in Figure 6-3.

Figure 6-3. State of Wait Queue (When rel_blk Is Issued)

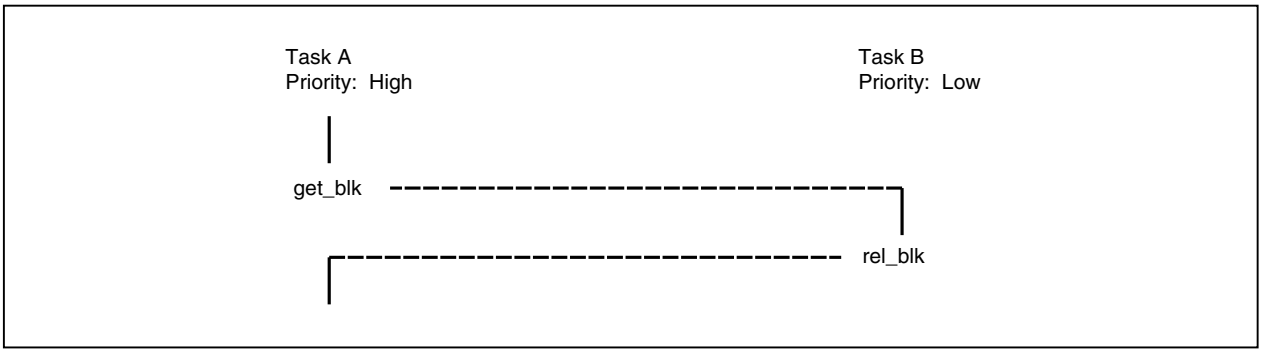


(4) The task A with the higher priority changes its state from ready to run.

Task B changes its state from run to ready.

Figure 6-4 shows the flow of dynamic use of memory by the memory pool explained in (1) through (4) above.

Figure 6-4. Dynamic Use of Memory by Memory Pool



CHAPTER 7 TIME MANAGEMENT FUNCTION

This chapter describes the time management function of the RX850 Pro.

7.1 Overview

Time management in the RX850 Pro is performed using clock interrupts which can be generated periodically by hardware (clock controller, etc.).

If a clock interrupt is issued, the RX850 Pro system clock processing is called and system clock update as well as processing related to time, such as delayed task wake-up, timeout, and starting of the cyclically activated handler, is executed.

7.2 System Clock

The system clock is a software timer that provides the time (in units of milliseconds, with a width of 48 bits) used for time management by the RX850 Pro.

The system clock is set to 0x0 at system initialization and updated in units of the basic clock cycle (specified at configuration) each time system clock processing is performed.

Caution The system clock managed by the RX850 Pro shall have a fixed width of 48 bits. The RX850 Pro ignores any overflow (that exceeding 48 bits) for the clock value.

7.2.1 Setting and reading the system clock

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

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

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

7.3 Timer Operations

Real-time processing requires functions synchronized with time (timer operation functions) such as stopping the processing of a certain task for a specific time and executing the processing of a handler for specific time. The RX850 Pro therefore provides the functions of delayed wake-up of a task, timeout, and starting of a cyclically activated handler, as timer operation functions.

7.4 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.

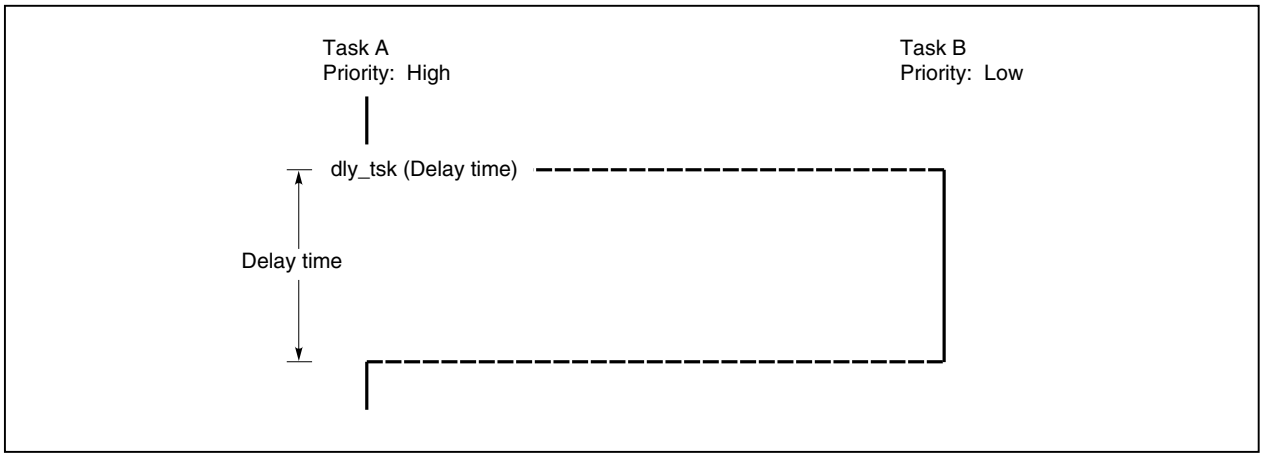
Delayed task wake-up is performed by issuing the dly_tsk system call.

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

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

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

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



7.5 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. The task then returns to the ready state.

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

- tslp_tsk system call

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

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

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

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

- twai_sem system call

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

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

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

- twai_flg system call

  The twai_flg system call checks whether the bit pattern is set so as to satisfy the wait condition required for the event flag specified by the 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 queued at the end of the wait queue of this event flag. Thus, the task leaves the run state and enters the wait state (the event flag wait state).

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

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

- trcv_msg system call

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

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

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

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

- tget_blk system call

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

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

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

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

7.6 Cyclically Activated Handler

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

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

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

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

7.6.1 Registering a cyclically activated handler

The RX850 Pro provides two interfaces for registering a cyclically activated handler. One enables static registration during system initialization (in the nucleus initialization section). The other enables dynamic registration by issuing a system call from within a processing program.

To register a cyclically activated handler with the RX850 Pro, an area in system memory is allocated for managing the cyclically activated handler (to be managed by the RX850 Pro), then initialized.

(1) Static registration of a cyclically activated handler

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

The RX850 Pro performs the processing for registering the cyclically activated handler, based on the information defined in the information file (including system information tables and system information header files) during system initialization. The cyclically activated handler is subsequently managed by the RX850 Pro.

(2) Dynamic registration of a cyclically activated handler

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

The RX850 Pro performs the processing for registering the cyclically activated handler, according to the information specified by the parameter when the def_cyc system call is issued.

The cyclically activated handler is subsequently managed by the RX850 Pro.

7.6.2 Activity state of cyclically activated handler

The activity state of a cyclically activated handler is used as a criterion for determining whether the RX850 Pro activated the cyclically activated handler.

The activity state is set when the cyclically activated handler is registered (during configuration or when the def_cyc system call is issued). However, the RX850 Pro allows the user to change the activity state of the cyclically activated handler from a user processing program.

- act_cyc system call
  Upon the issuance of the act_cyc system call, the activity state of the cyclically activated handler is switched ON/OFF, as specified by the parameter.

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

While the RX850 Pro is running, the cycle counter continues to count even when the activity state of the cyclically activated handler is OFF. In some cases, when an act_cyc system call is issued to switch the activity state of the cyclically activated handler from OFF to ON, the first restart request could be issued sooner than the activation time interval specified when it was registered (during configuration or upon the issuance of the def_cyc system call). To prevent this, the user must specify TCY_INI to initialize the cycle counter as well as TCY_ON to restart the cyclically activated handler when issuing the act_cyc system call. The first restart request will then be issued in sync with the time interval, specified when it was registered.

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

In those figures, ΔT indicates the activation time interval specified upon the registration of the cyclically activated handler. In addition, the relationship between Δt and ΔT in Figure 7-2 is assumed to be $\Delta t \leq \Delta T$.

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

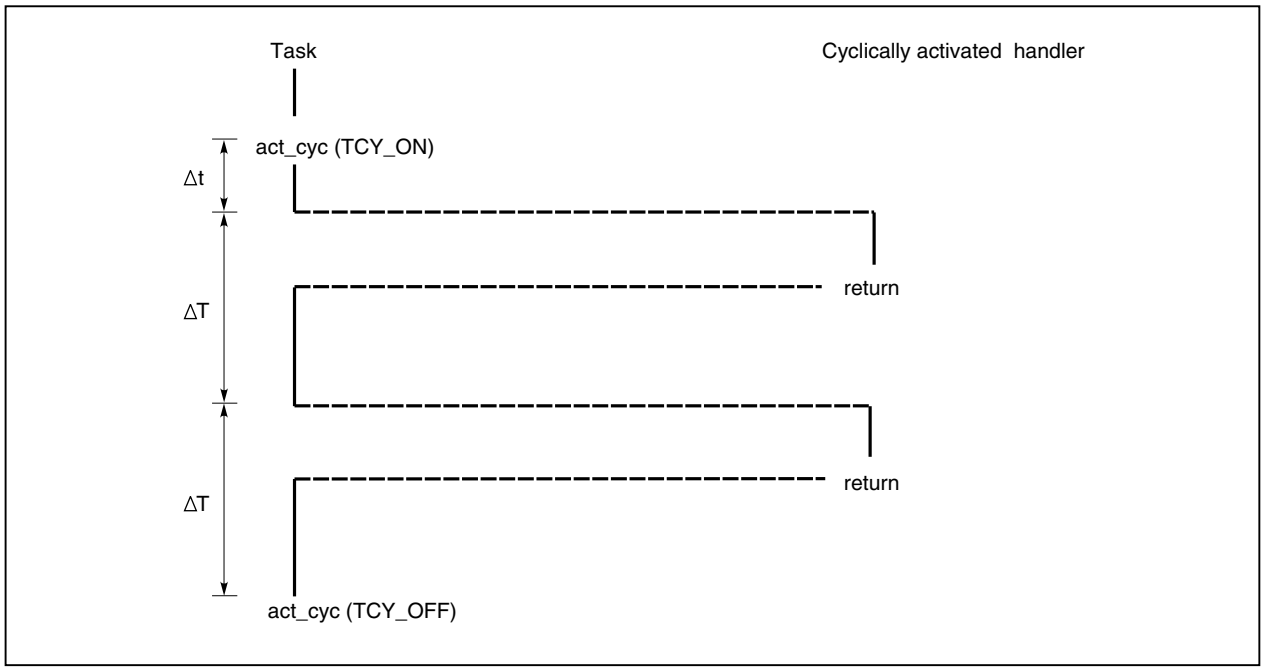
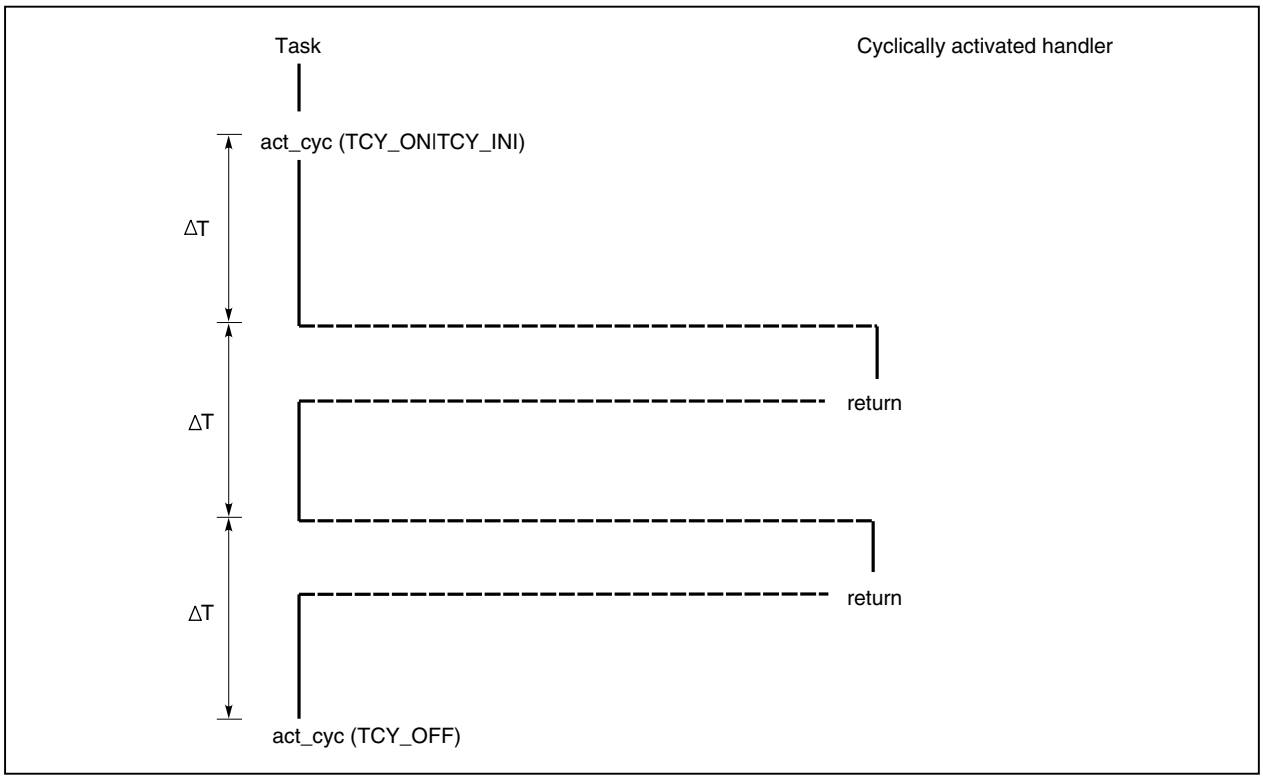


Figure 7-3. Flow of Processing After Issuance of act_cyc (TCY_ON|TCY_INI)



7.6.3 Internal processing performed by cyclically activated handler

After the occurrence of a timer interrupt, the RX850 Pro performs preprocessing for interruption before control is passed to the cyclically activated handler. When control is returned from the cyclically activated handler, the RX850 Pro performs interrupt postprocessing.

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

(1) Saving/restoring the registers

Based on the function call protocol for the C compiler (CA850 or CCV850), the RX850 Pro saves the work registers when control is passed to the cyclically activated handler, and restores them upon the return of control from the handler. Therefore, the cyclically activated handler does not have to save the work registers when it starts, nor restore them upon the completion of its processing. Save/restoration of the registers should not be coded in the description of the cyclically activated handler.

(2) Stack switching

The RX850 Pro performs stack switching when control is passed to the cyclically activated handler and upon a return from the handler. Therefore, the cyclically activated handler does not have to switch to the interrupt handler stack when it starts, nor switch to the original stack upon the completion of its processing. However, if the interrupt handler stack is not defined during configuration, stack switching is not performed and system continues to use that stack being used upon the occurrence of an interrupt.

(3) Limitations imposed on system calls

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

- **Task management system calls**

sta_tsk	chg_pri	rot_rdq	rel_wai	get_tid
ref_tsk	vget_tid			

- **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	vget_sid	set_flg
clr_flg	pol_flg	ref_flg	vget_fid	snd_msg
prcv_msg	ref_mbx	vget_mid		

- **Interrupt management system calls**

def_int	ena_int	dis_int	chg_icr	ref_icr

- **Memory pool management system calls**

pget_blk	rel_blk	ref_mpl	vget_pid

- **Time management system calls**

set_tim	get_tim	def_cyc	act_cyc	ref_cyc

- **System management system calls**

get_ver	ref_sys	def_svc	viss_svc

(4) Return processing from the cyclically activated handler

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

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

7.6.4 Acquiring cyclically activated handler information

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

- ref_cyc system call

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

  The cyclically activated handler information consists of the following:

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

⋆ 7.6.5 Interrupts in cyclically activated handler

The cyclically activated handler starts processing in the interrupt enabled state. To disable interrupts during use of the cyclically activated handler, disable interrupts at the start of the handler.

Since the RX850 Pro provides two types of nucleus common parts (rxcore.o and rxtmcore.o), the interrupts that can be acknowledged within the cyclically activated handler differ depending on the nucleus common part used.

- **When rxcore.o is used**
  Although the cyclically activated handler is called from the timer handler, all levels of interrupts can be acknowledged because the interrupt processing is performed during timer handler execution.

- **When rxtmcore.o is used**
  Although the cyclically activated handler is called from the timer handler, only the interrupts with a higher priority than timer interrupts can be acknowledged because the interrupt processing is not performed during timer handler execution. In addition, since timer interrupts are held pending even when interrupts are enabled, to execute a time-consuming processing within the cyclically activated handler, caution is required because displacement may occur between the time that has actually elapsed and the time managed by the RX850 Pro.

Because the cyclically activated handler is developed as an indirectly activated interrupt handler, it operates on the handler stack at execution.

7.6.6 Activation order of cyclically activated handler

When multiple cyclically activated handlers for which the activation interval time simultaneously elapsed exist, activated in order from the handler with shorter activation interval time specified. In addition, if the activation interval time elapsed during the another cyclically activated handler execution, the cyclically activated handler is not activated immediately but activated after the end of the cyclically activated handler currently executed.

CHAPTER 8 SCHEDULER

This chapter explains the task scheduling performed by the RX850 Pro.

8.1 Overview

By monitoring the dynamically changing task states, the RX850 Pro 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 Pro scheduler uses an event-driven technique, in which the scheduler operates in response to the occurrence of some event.

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

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

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

- **Task management system calls**

  sta_tsk ext_tsk exd_tsk ena_dsp chg_pri

  rot_rdq rel_wai

- **Task-associated synchronization system calls**

  rsm_tsk frsm_tsk slp_tsk tslp_tsk wup_tsk

- **Synchronous communication system calls**

  del_sem sig_sem wai_sem twai_sem del_flg

  set_flg wai_flg twai_flg del_mbx snd_msg

  rcv_msg trcv_msg

- **Interrupt management system calls**

  ret_int ret_wup vret_clk unl_cpu

- **Memory pool management system calls**

  del_mpl get_blk tget_blk rel_blk

- **Time management system call**

  dly_tsk

8.3 Scheduling Method

The RX850 Pro 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.

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

8.3.2 FCFS method

The RX850 Pro 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 (the task that has been in the ready state for the longest time) and assigns a processing time to the selected task.

8.4 Implementing a Round-Robin Method

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

The RX850 Pro provides system calls such as rot_rdq to implement a scheduling method (round-robin method) that can overcome the problems incurred by the priority and FCFS methods.

The round-robin method can be implemented as follows:

Conditions

- Task priority
  Task A = Task B = Task C

- State of tasks
  Task A: Run state
  Task B: Ready state
  Task C: Ready state

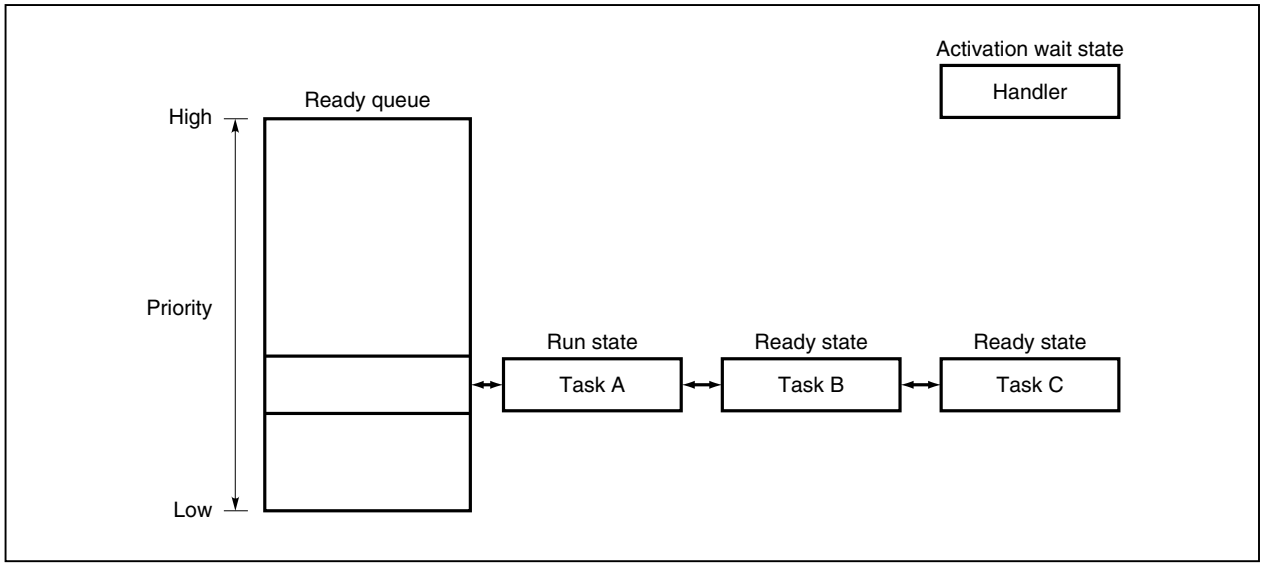
- Cyclically activated handler attributes
  Activity state: ON
  Activation interval: ΔT (unit: Basic clock cycle)
  Processing: Rotation of the ready queues (issue of the rot_rdq system call)

(1) Task A is currently running.

The other tasks (B and C) have the same priority as task A, but they cannot be executed unless task A enters another state or relinquishes control of the processor.

The ready queue becomes as shown in Figure 8-1.

Figure 8-1. Ready Queue State (1)

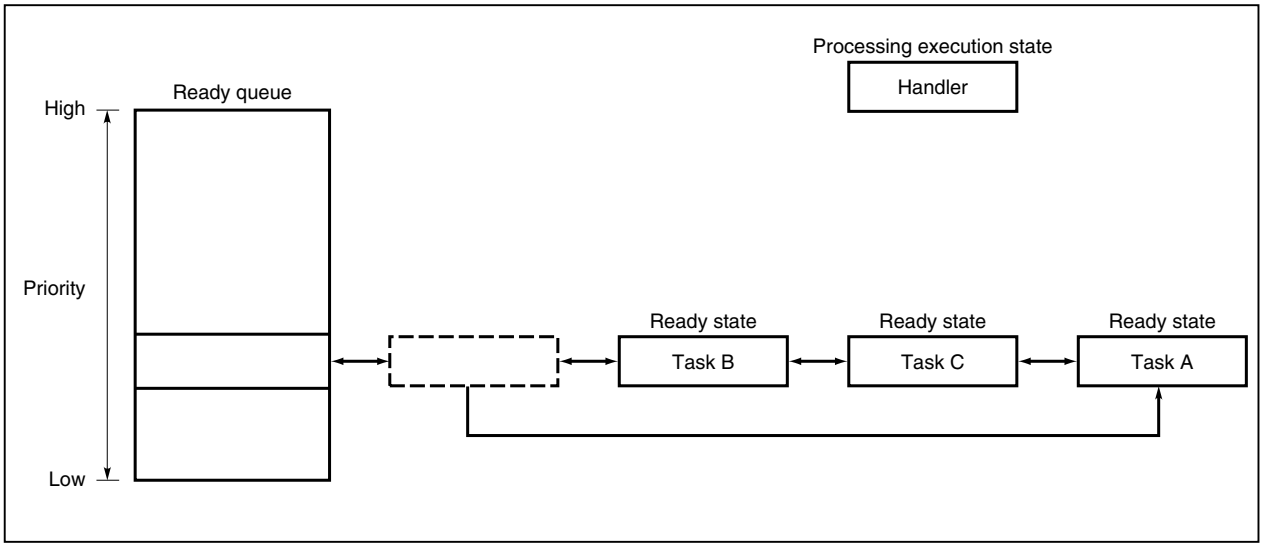


(2) Cyclically activated handler starts when the predetermined period of time has passed and issues the rot_rdq system call.

In this way, task A is queued at the tail end of the ready queue in accordance with its priority level and changes from the run state to ready state.

The ready queue changes to the state shown in Figure 8-2.

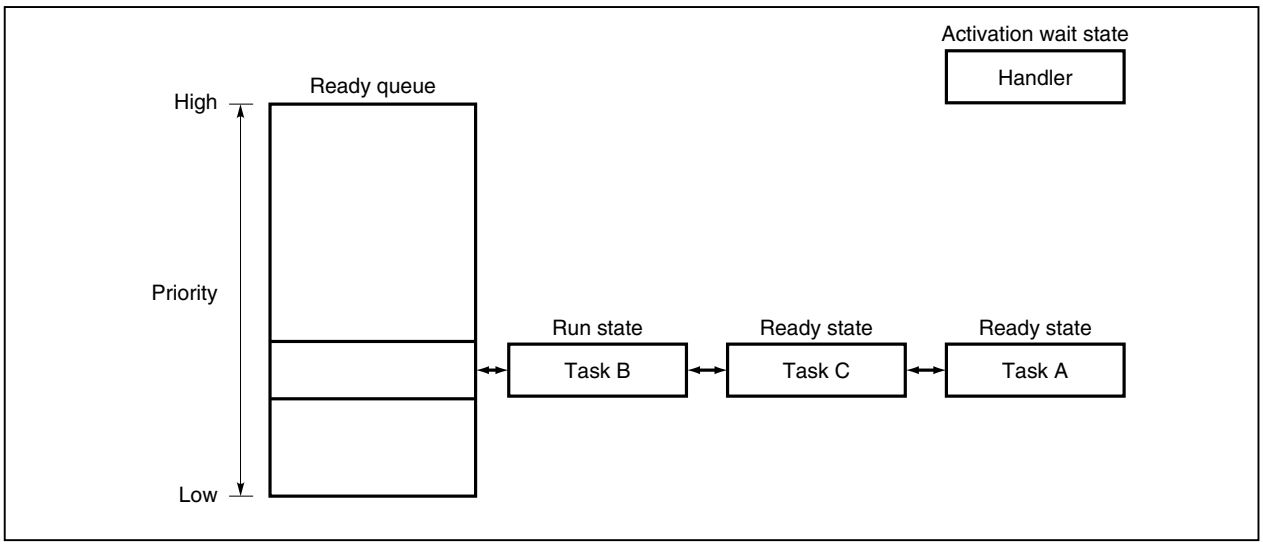
Figure 8-2. Ready Queue State (2)



(3) Task A changes from the run state to the ready state and task B changes from the ready state to the run state.

Figure 8-3 shows the ready queue state at this time.

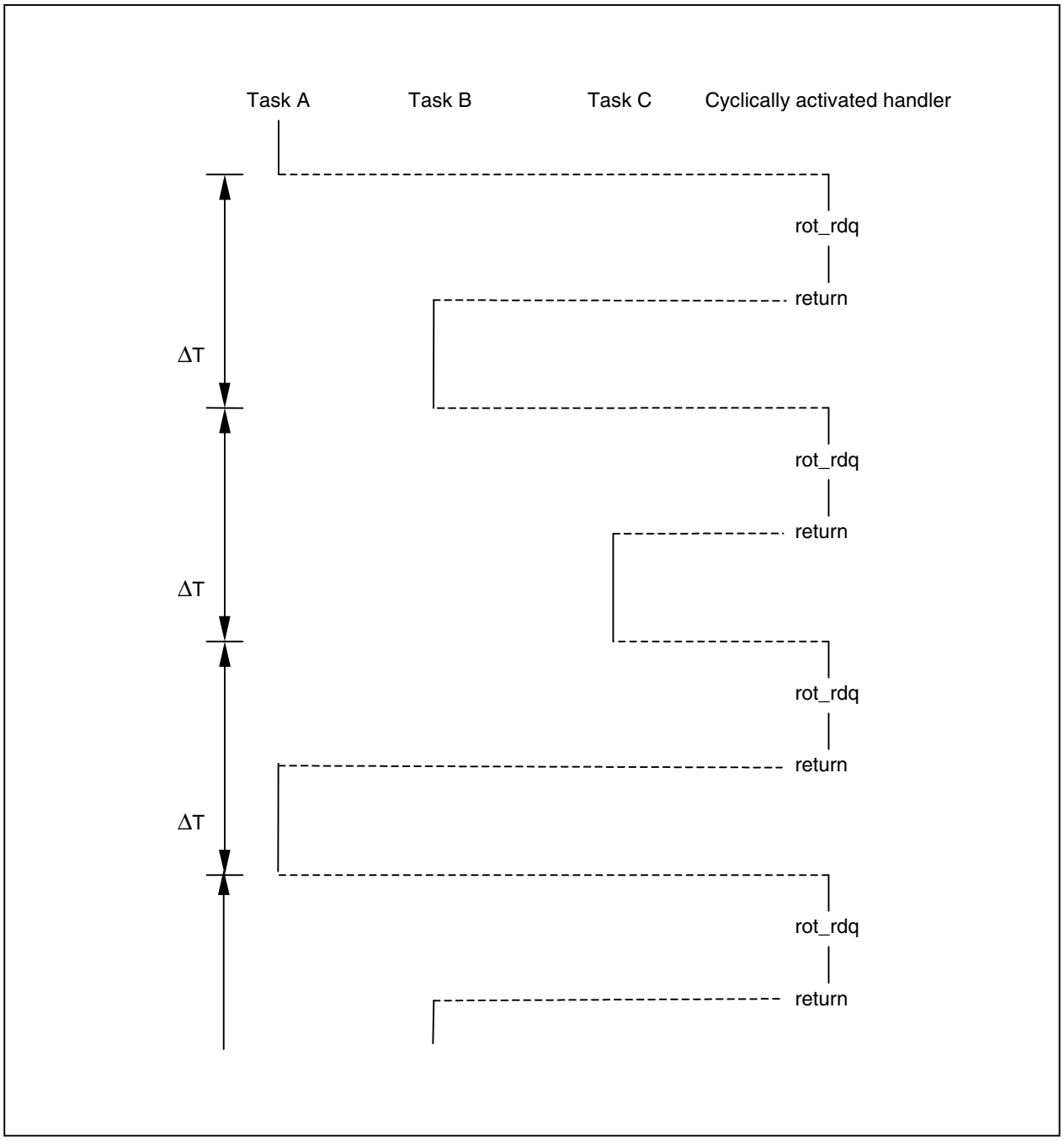
Figure 8-3. Ready Queue State (3)



(4) By issuing the rot_rdq system call from the cyclically activated handler, which is started at constant intervals, the 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. Flow of Processing by Using Round-Robin Method



8.5 Scheduling Lock Function

In the RX850 Pro a function is offered which drives the scheduler from a user processing program (task) and which disables or resumes dispatch processing (task scheduling processing).

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

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

- ena_dsp system call
  Resumes dispatching (task scheduling).
  When the dis_dsp system call has been issued, if a system call that requires task scheduling (such as chg_pri or sig_sem) is issued, the RX850 Pro merely executes processing such as wait queue operation until the ena_dsp system call is issued. Actual scheduling is delayed and batch-executed upon the issuance of the ena_dsp system call.

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

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

The flow of control if scheduling processing is not delayed (normal) is shown in Figure 8-5 and the flow of control if the dis_dsp and loc_cpu system calls are issued is shown in Figure 8-6 and Figure 8-7.

Figure 8-5. Flow of Control if Scheduling Processing Is Not Delayed (Normal)

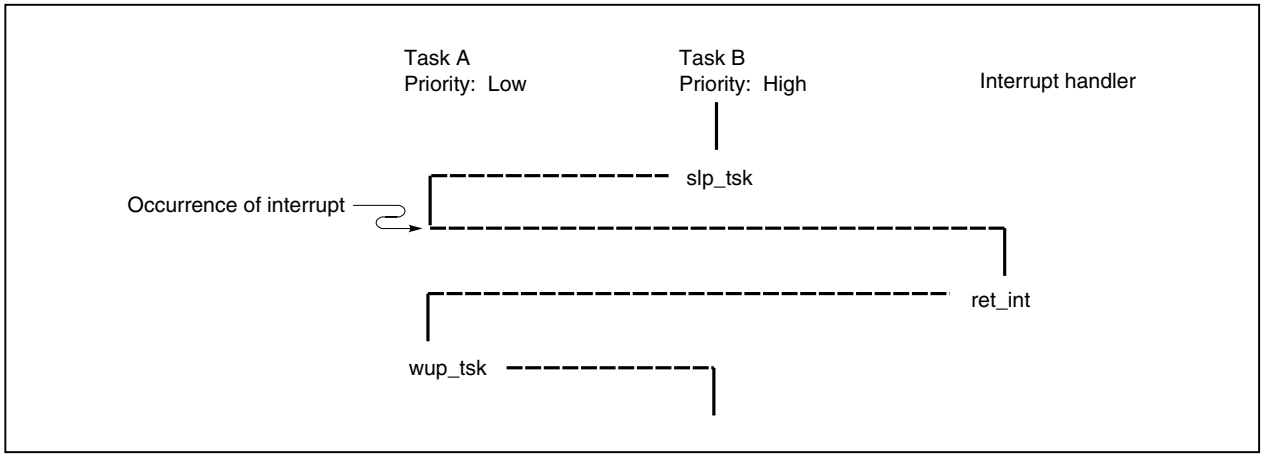


Figure 8-6. Flow of Control if dis_dsp System Call Is Issued

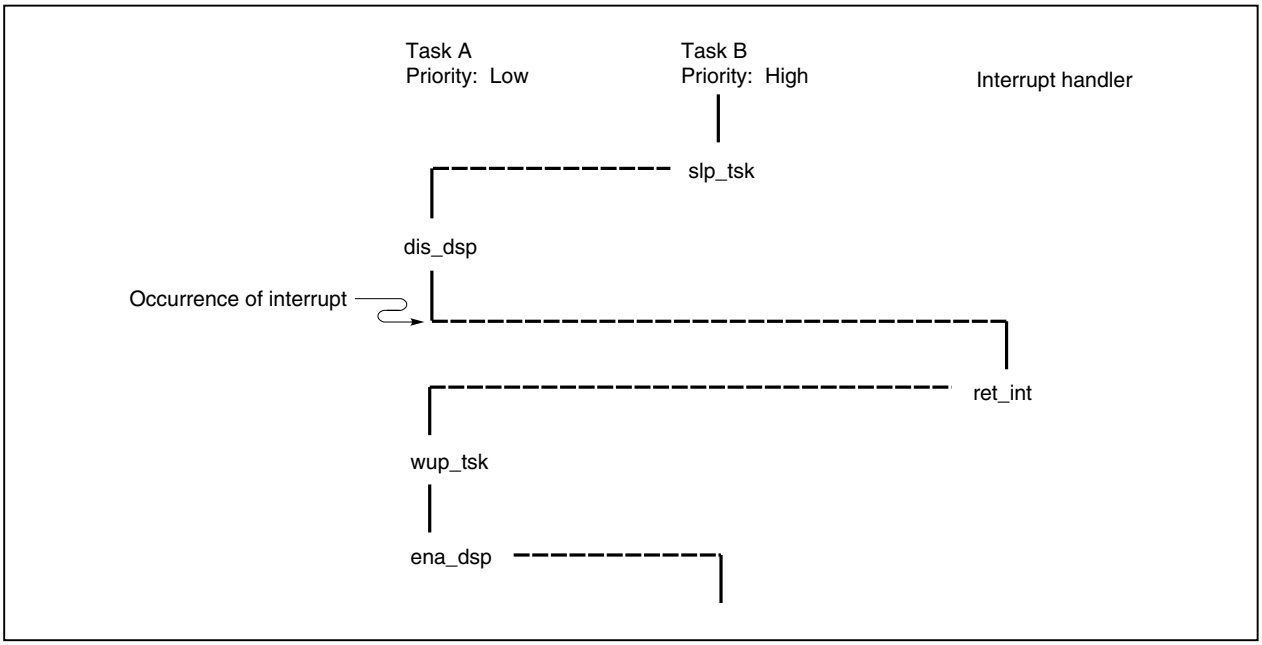
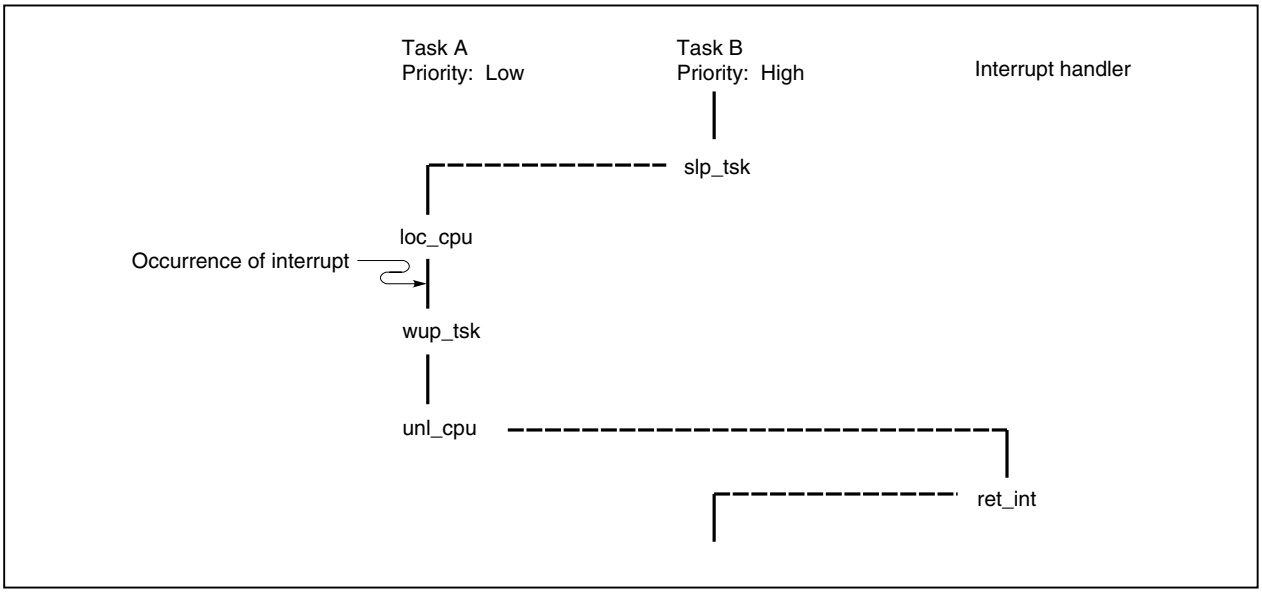


Figure 8-7. Flow of Control if loc_cpu System Call Is Issued



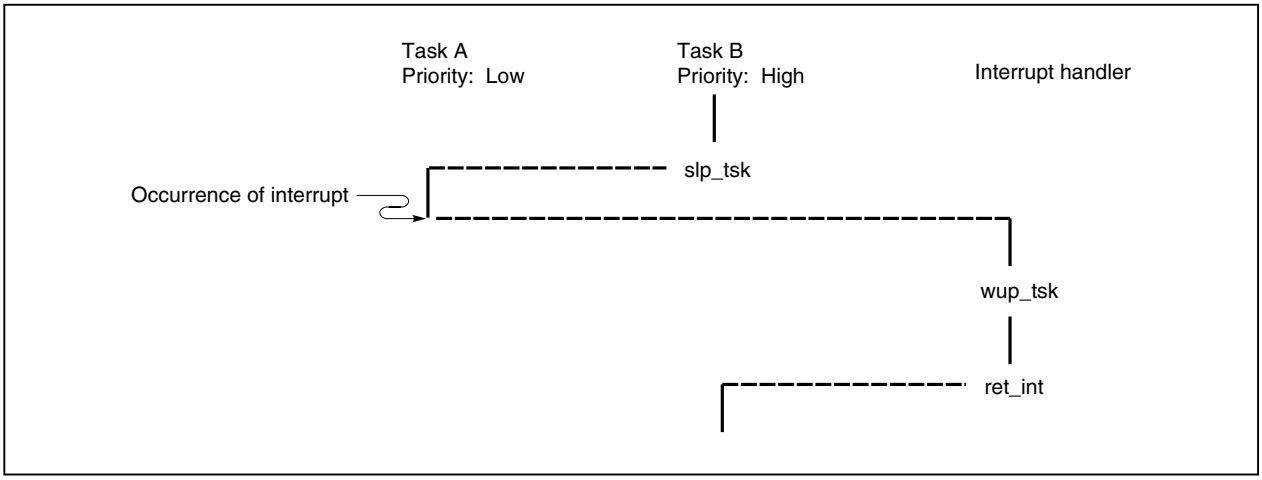
8.6 Scheduling While Handler Is Operating

To quickly terminate handlers (interrupt handlers and cyclically activated handlers), the RX850 Pro 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 Pro merely executes processing such as wait queue operation until the completion of return processing from the handler (such as ret_int system call or the issue of return instruction). Actual scheduling is delayed and then batch-executed 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. Flow of Control if wup_tsk System Call Is Issued



8.7 Idle Handler

8.7.1 Idle handler

The idle handler is started from the scheduler if all the tasks (user defined tasks) are not in the run state or not in the ready state, that is, if there is not even one task which is an object of RX850 Pro scheduling in the system.

The processing of the idle handler is to switch the CPU to the HALT state. Therefore, if there is not even one task in the system, the RX850 Pro switches the CPU to the HALT state.

However, this idle handler cannot switch the CPU to the IDLE or STOP state. To switch to the IDLE or STOP state, or to describe idle processing, create a task with the lowest priority and use it as an idle task. This realizes processing identical to the idle handler. However, since the HALT, IDLE, or STOP state is released by an interrupt, be sure not to leave interrupts in a disabled state in the idle task.

CHAPTER 9 SYSTEM INITIALIZATION

This chapter explains the system initialization performed by the RX850 Pro.

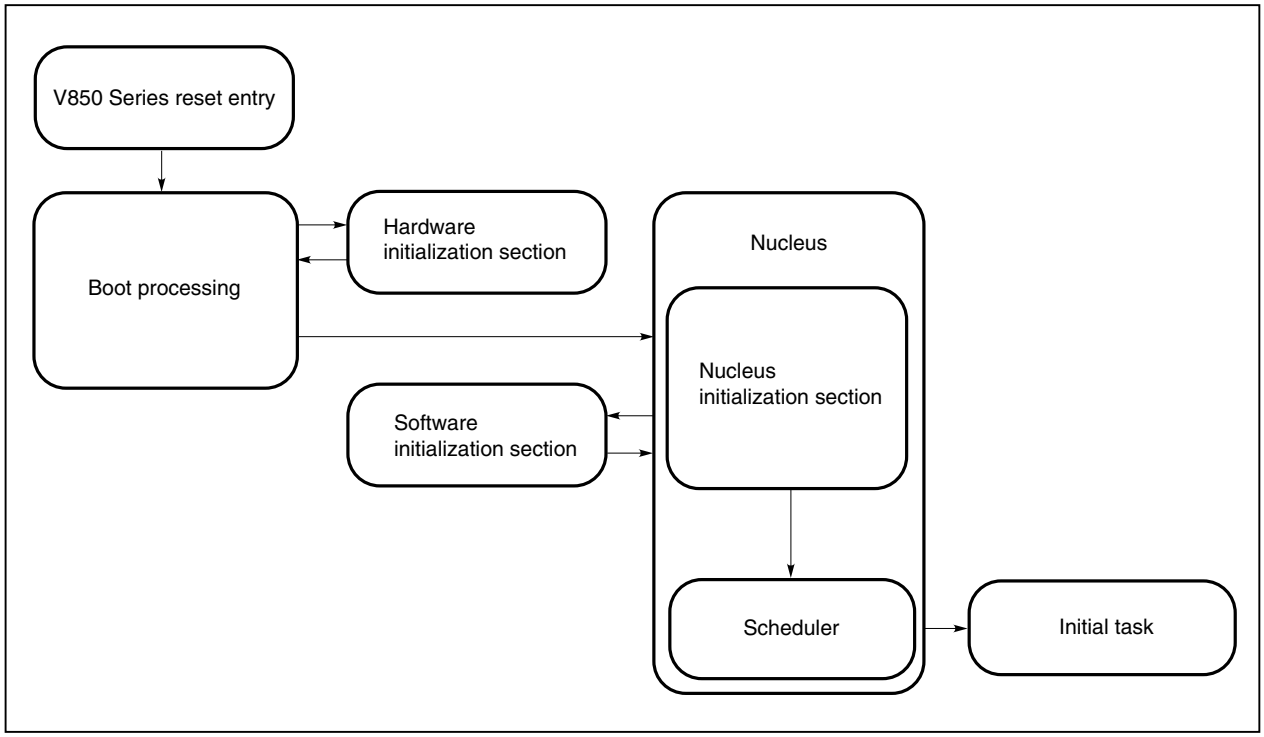
For details of the system initialization, refer to the **RX850 Pro Installation User's Manual (U13774E)**.

9.1 Overview

System initialization consists of initializing the hardware required by the RX850 Pro, as well as initializing the software. In other words, in the RX850 Pro, 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



9.2 Boot Processing

Boot processing is the function assigned to the V850 Series reset entry (handler address: 0x0) and the first function executed in system initialization. The files boot.s (NEC Electronics version) and boot.850 (GHS version) are used in the sample boot processing (function name: _ _boot).

Boot processing involves the following.

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

In the sample boot processing, the processing can be rewritten to adapt to user needs.

9.3 Hardware Initialization Section

The hardware initialization section is a function called from the boot processing and it is prepared for initializing the hardware in the execution environment (target system). The file init.c is used in the sample initialization (function name: reset).

In this hardware initialization section, the following processing is performed.

- Initialization of the internal unit
  - Initialization of an interrupt controller
  - Initialization of a clock controller
- Initialization of a peripheral controller
- Returns control to boot processing

The hardware initialization section depends on the hardware configuration of the execution environment. Designing this section into the LSI improves portability to the target system and simplifies customization. Rewrite in accordance with the user execution environment.

9.4 Nucleus Initialization Section

The nucleus initialization section is a function called after the boot processing completion and it generates and initializes the management objects based on the information (such as task information or semaphore information) described in the information files (system information table and system information header file). The RX850 Pro is activated after completion of this processing. This processing section is included in the nucleus library.

The nucleus initialization section performs the following processing.

- Generation/initialization of management objects
  - Task generation
  - Generation/initialization of a semaphore
  - Generation/initialization of event flags
  - Generation/initialization of a memory pool
  - Registration of the indirectly activated interrupt handler
  - Registration of the cyclically activated handler
  - Registration of the extended SVC handler
- Activation of an initial task
- Activation of the system task (idle task)
- Calling of the software initialization section
- Transfer of control to the scheduler

9.5 Software Initialization Section

The software initialization section is a function called from the nucleus initialization section and used if some processing is to be executed before the activation of the RX850 Pro. The file varfunc.c is used in the sample processing (function name: varfunc).

The software initialization section performs the following processing.

- Copying of an initialization data
- Returns control to the nucleus initialization section

CHAPTER 10 INTERFACE LIBRARY

This chapter explains the interface library.

For details of the interface library, refer to the **RX850 Pro Installation User's Manual (U13774E)**.

10.1 Overview

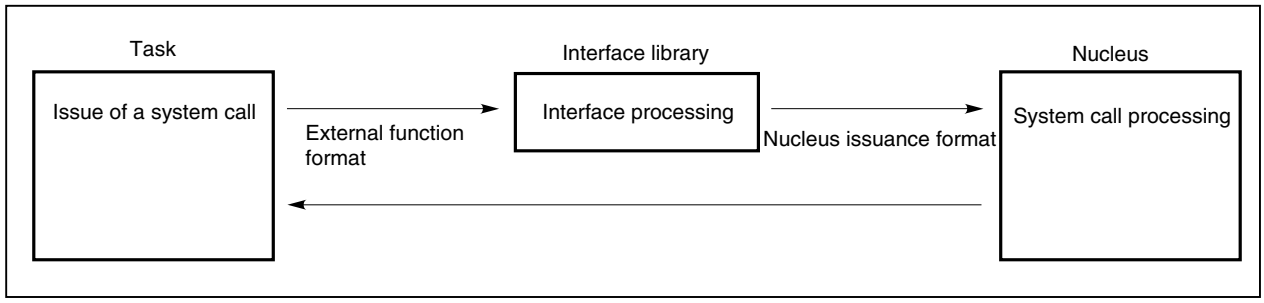
In the RX850 Pro, an interface library is provided which is positioned midway between the user processing program and the RX850 Pro nucleus. The interface library has a function for transferring control after performing setting of each type of necessary information, etc. for enabling processing by the nucleus.

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

It is therefore necessary to convert the system call issue format or expanded SVC handler calling format from the external function format to the nucleus issuance format (interfacing). There is an interface which performs the role of intermediary between the processing program and the nucleus for each system call. All these interfaces collected together are called the interface library.

Figure 10-1 shows the positioning of the interface library in the RX850 Pro.

Figure 10-1. Positioning of Interface Library



10.2 Processing in the Interface Library

The following processing is performed in the interface library.

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

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

10.3 Types of Interface Libraries

There are two types of interface libraries offered with RX850 Pro, one with a function for checking system call parameters, and one without this function. The type of interface library which will be incorporated is specified at linking.

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

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

Remark Errors in which return values are returned with the library which does not have a parameter check function are marked by "*" in the system call return value column in **CHAPTER 11 SYSTEM CALLS**.

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

10.4 Supported Interface Libraries

The RX850 Pro supports the following two interface libraries:

- For NEC Electronics V850 Series C compiler CA850
- For C cross V800 compiler CCV850 manufactured by Green Hills Software, Inc.

Remark To use other compilers, the interface library must be rewritten in accordance with the register used in its compiler. For the addresses where the interface libraries are stored, refer to the **RX850 Pro Installation User's Manual (U13774E)**.

CHAPTER 11 SYSTEM CALLS

This chapter describes the system calls supported by the RX850 Pro.

11.1 Overview

A system call is a procedure or function for invoking RX850 Pro service routines from the user's processing programs (tasks/non-tasks). The user can use system calls to indirectly manipulate those resources (such as counters and queues) that are managed directly by the RX850 Pro.

The RX850 Pro supports its own seven system calls as well as the 65 defined in the μITRON3.0 specifications, thus enhancing the versatility of application systems.

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

(1) Task management system calls (14)

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

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

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

(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 suspended task, a function for placing a task in the wake-up wait state and waking up a task currently in the wake-up wait state, and another function for canceling a task wake-up request.

sus_tsk	rsm_tsk	frsm_tsk	slp_tsk	tslp_tsk
wup_tsk	can_wup			

(3) Synchronous communication system calls (25)

These system calls are used for the synchronization (exclusive control and queuing) and communication between tasks.

This group provides a function for manipulating semaphores, a function for manipulating events and flags, and a function for manipulating mailboxes.

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

(4) Interrupt management system calls (9)

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

This group provides a function for registering an indirectly activated interrupt handler and subsequently canceling the registration, a function for returning from a directly activated interrupt handler, and a function for changing or referencing an interrupt-enabled level.

def_int	ret_int	ret_wup	ena_int	dis_int
loc_cpu	unl_cpu	chg_icr	ref_icr	

(5) Memory pool management system calls (8)

These system calls allocate memory.

This group provides a function for creating and deleting a memory pool, a function for acquiring and returning a memory block, and a function for referencing the status of a memory pool.

cre_mpl	del_mpl	get_blk	pget_blk	tget_blk
rel_blk	ref_mpl	vget_pid		

(6) Time management system calls (6)

These system calls perform processing that is dependent on time.

This group provides a function for setting or referencing the system clock, a function for placing a task in the timeout wait state, a function for registering a cyclically activated handler and subsequently canceling the registration, and a function for controlling and referencing the state of a cyclically activated handler.

set_tim	get_tim	dly_tsk	def_cyc	act_cyc
ref_cyc				

(7) System management system calls (4)

These system calls perform processing that varies with the system.

This group provides a function for acquiring version information, a function for referencing the system status, a function for registering an extended SVC handler and subsequently canceling the registration, and a function for calling an extended SVC handler.

get_ver	ref_sys	def_svc	viss_svc

11.2 Calling System Calls

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

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

Caution The RX850 Pro declares the prototype of a system call in the stdrx85p.h file. Accordingly, when issuing a system call from a processing program, the following must be coded to include the header file:

```
#include <stdrx85p.h>
```

11.3 System Call Function Codes

The system calls supported by the RX850 Pro are assigned function codes conforming to the μITRON3.0 specifications.

Table 11-1 lists the function codes assigned to system calls.

In the RX850 Pro, a value of 1 or greater is used when registering an extended SVC handler described by the user.

Table 11-1. System Call Function Codes

Function Code	Classification
–256 to –225	RX850 Pro original system calls
–224 to –5	System calls conforming to the μITRON3.0 specifications
–4 to 0	Reserved by the system
1 or more	Extended SVC handler

11.4 Data Types of Parameters

The system calls supported by the RX850 Pro have parameters that are defined based on data types that conform to the μITRON3.0 specifications.

Table 11-2 lists the data types of the parameters specified upon the issuance of a system call.

Table 11-2. Data Types of Parameters

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	Processing program start address
BOOL	short	Boolean value
FN	short	Function code
ID	short	Object ID number
BOOL_ID	short	Wait task available or not
HNO	short	Cyclically activated handler specification number
ATR	unsigned short	Object attribute
ER	long	Error code
PRI	short	Task priority
TMO	long	Wait time
CYCTIME	long	Cyclically activated time interval (residual time)
DLYTIME	long	Delay time

11.5 Parameter Value Range

Some of the system call parameters supported by the RX850 Pro have a range of permissible values, while others allow the use of only system reserved specific values.

Table 11-3 lists the ranges of parameter values that can be specified upon the issuance of a system call.

Table 11-3. Ranges of Parameter Values

Parameter Type	Value Range
Object ID number	0x0 to max_cnt[Note 1]
Object key ID number	–0x8000 to 0x7FFF[Note 2]
Interrupt handler interrupt level	0x0 to 0xF
Specification number of cyclically activated handler	0x1 to max_cnt
Extended function code of extended SVC handler	0x1 to max_cnt
Object priority	0x1 to max_cnt
Maximum number of semaphore resources	0x1 to max_cnt
Interrupt enable level of maskable interrupt	0x0 to 0xF
System clock time	0x0 to 0x7FFF FFFF FFFF
Wait time	–0x1 to 0x7FFF FFFF
Delay time	0x0 to 0x7FFF FFFF
Activation time interval of cyclically activated handler	0x1 to 0x7FFF FFFF
Task stack size	0x0 to 0x7FFF FFFF
Memory pool size	0x1 to 0x7FFF FFFF
Memory block size	0x1 to 0x7FFF FFFF
Message priority level	0x1 to 0x7FFF

Notes 1. max_cnt: Maximum number of objects specified during system configuration

2. “0x0” cannot be specified for the object key ID number.

11.6 System Call Return Values

The system call return values supported by the RX850 Pro are based on the μITRON3.0 specifications. Table 11-4 lists the system call return values.

Table 11-4. System Call Return Values

Macro	Value	Description
E_OK	0	Normal termination
E_NOMEM	–10	An area for objects cannot be allocated.
E_NOSPT	–17	A system call with the CF not defined, or an unregistered extended SVC handler was called.
E_RSATR	–24	Invalid object attribute specification
E_PAR	–33	Invalid parameter specification
E_ID	–35	Invalid ID number specification
E_NOEXS	–52	No relevant object exists.
E_OBJ	–63	The status of the specified object is invalid.
E_OACV	–66	An unauthorized ID number was specified.
E_CTX	–69	The state in which the system call is issued is invalid.
E_QOVR	–73	The count exceeded 127.
E_DLT	–81	The target object was deleted.
E_TMOUT	–85	Timeout
E_RLWAI	–86	A wait state was forcibly canceled by the rel_wai system call.

11.7 System Call Extension

The RX850 Pro supports the extension of system calls (functions coded by users are registered in the nucleus as extended system calls).

No limitations are imposed on those functions registered as extended system calls; standard system calls (system calls supported by the RX850 Pro) can also be included. If, however, standard system calls that can be issued only in the task state are included, the issuance state of the extended system calls is limited to “issuable only from task.”

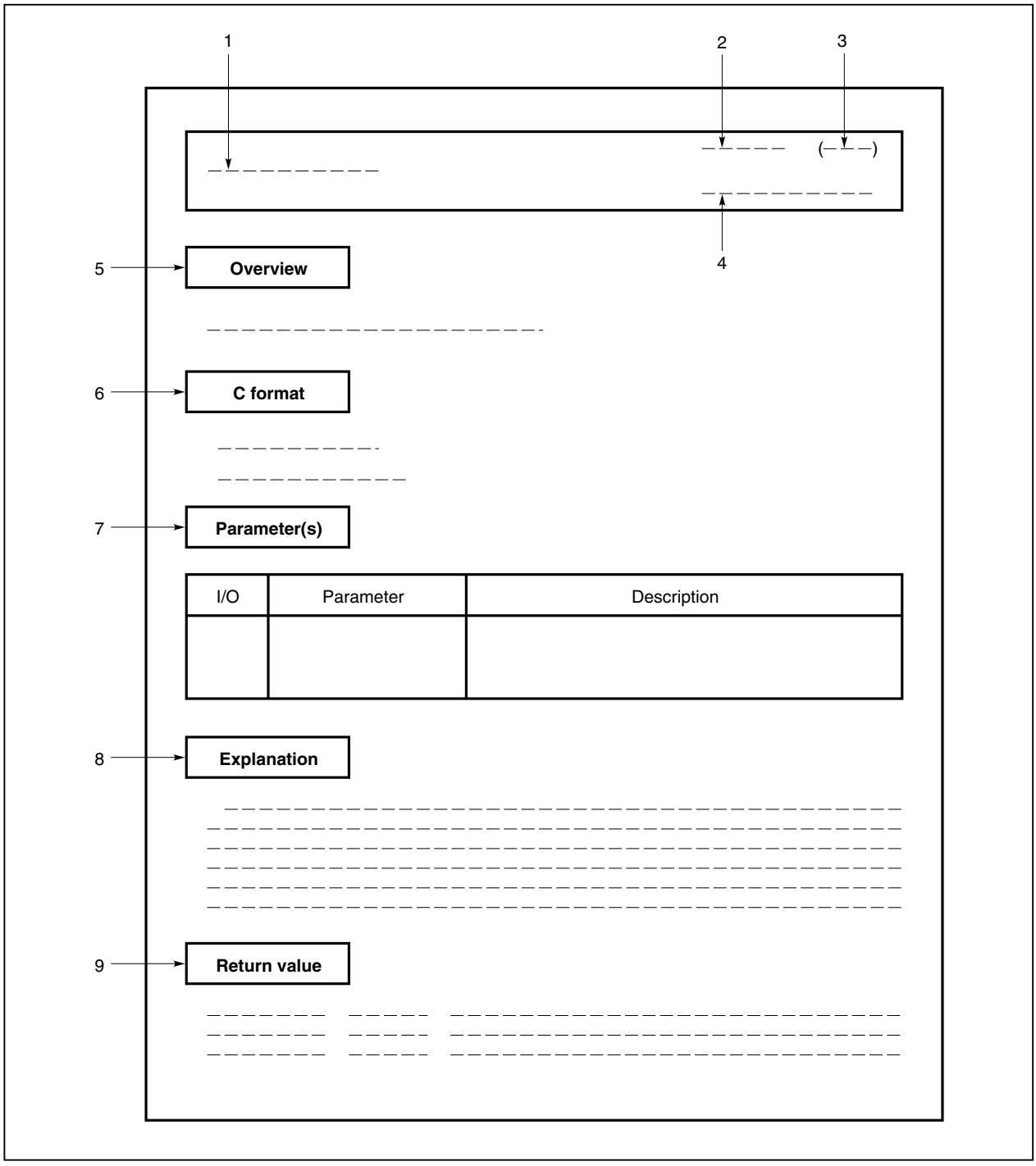
Extended system calls are positioned as user-defined system calls, despite their having properties similar to tasks. That is, like standard system calls, the scheduler is started upon the termination of processing and an optimum task is selected.

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

11.8 Explanation of System Calls

The following explains the system calls supported by the RX850 Pro, in the format shown below.

Figure 11-1. System Call Description Format



(1) Name

Indicates the name of the system call.

(2) Semantics

Indicates the source of the name of the system call.

(3) Function code

Indicates the function code of the system call.

(4) Origin of system call

Indicates where the system call can be issued.

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

(5) Overview

Outlines the functions of the system call.

(6) C format

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

(7) Parameter(s)

System call parameters are explained in the following format.

I/O	Parameter	Description
A	B	C

A: Parameter classification
- I ... Parameter input to RX850 Pro
- O ... Parameter output from RX850 Pro

B: Parameter data type

C: Description of parameter

(8) Explanation

Explains the function of a system call.

(9) Return value

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

Return value marked with an asterisk (*):	Value returned by both RX850 Pro having and that not having the parameter check function
Return value not marked with an asterisk (*):	Value returned only by RX850 Pro having the parameter check function

11.8.1 Task management system calls

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

Table 11-5 lists the task management system calls.

Table 11-5. Task Management System Calls

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

cre_tsk

Create Task (–17)

Task

Overview

Creates a task.

C format

- When an ID number is specified

```
#include      <stdrx85p.h>
ER            ercd = cre_tsk(ID tskid, T_CTSK *pk_ctsk);
```

- When an ID number is not specified

```
#include      <stdrx85p.h>
ER            ercd = cre_tsk(ID_AUTO, T_CTSK *pk_ctsk, ID *p_tskid);
```

Parameters

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

- Structure of task creation information T_CTSK

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

Explanation

The RX850 Pro supports two types of interfaces for task creation: one in which an ID number is specified for task creation, and another in which an ID number is not specified.

- When an ID number is specified
  A task having the ID number specified by *tskid* is created based on the information specified by *pk_ctsk*.
  The specified task changes from the non-existent state to the dormant state, in which it is managed by the RX850 Pro.

- When an ID number is not specified
  A task is created based on the information specified by *pk_ctsk*.
  The specified task changes from the non-existent state to the dormant state, in which it is managed by the RX850 Pro.
  An ID number is allocated by the RX850 Pro and the allocated ID number is stored in the area specified by *p_tskid*.

The following describes task creation information in detail.

exinf ... Extended information

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

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

tskatr ... Task attribute

Bit 0 .. Task language

TA_ASM(0): Assembly language

TA_HLNG(1): C language

Bit 8 .. Existence of key ID number specification

TA_KEYID(1): Specifies key ID number

Bit 9 .. Memory area specification

TA_SPOL0(0): Secures the stack area from system memory area 0.

TA_SPOL1(1): Secures the stack area from system memory area 1.

Bit 10 .. Existence of specific GP register value specification

TA_DPID(1): Specifies a specific GP register value.

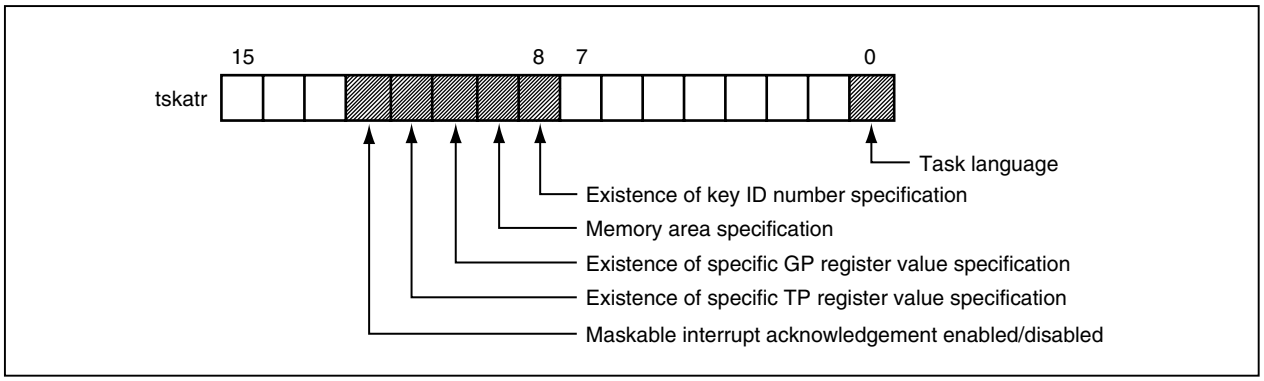
Bit 11 .. Existence of specific TP register value specification

TA_DPIC(1): Specifies a specific TP register value.

Bit 12 .. Maskable interrupt acknowledgement enabled or disabled

TA_ENAINT(0): When a task is activated, the acknowledgement of maskable interrupts is enabled.

TA_DISINT(1): When a task is activated, the acknowledgement of maskable interrupts is disabled.



task ... Task activation address
itskpri ... Task initial priority (assigned upon activation)
stksz ... Stack size of task (unit: bytes)
gp ... Specific GP register value for task
tp ... Specific TP register value for task
keyid ... Task key ID number

Remark If the value of Bit 8 is not 1 (TA_KEYID), the contents of keyid are meaningless.
If the value of Bit 10 is not 1 (TA_DPID), the contents of gp are meaningless.
If the value of Bit 11 is not 1 (TA_DPIC), the contents of tp are meaningless.

Return value

*E_OK	0	Normal termination
*E_NOMEM	–10	An area for task management block cannot be allocated.
*E_NOSPT	–17	The cre_tsk system call is not defined as CF.
E_RSATR	–24	Invalid specification of attribute tskatr
E_PAR	–33	Invalid parameter specification • The start address of the packet storing task creation information is invalid (*pk_ctsk* = 0). • Invalid activation address specification (task = 0) • Invalid initial priority specification (itskpri ≤ 0, maximum priority < itskpri) • Invalid key ID number specification (keyid = 0) (at TA_KEYID attribute specification) • The address of the area used to store the ID number is invalid (*p_tskid* = 0) (When a task is created with no ID number specified)
E_ID	–35	Invalid ID number specification (maximum number of tasks created < *tskid*)
*E_OBJ	–63	A task having the specified ID number has already been created.
E_OACV	–66	An unauthorized ID number (*tskid* ≤ 0) was specified.
E_CTX	–69	The cre_tsk system call was issued from a non-task.

del_tsk

Delete Task (–18)

Task

Overview

Deletes another task.

C format

```
#include      <stdrx85p.h>
ER            ercd = del_tsk(ID tskid);
```

Parameter

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

Explanation

This system call changes the task specified by *tskid* from the dormant state to the non-existent state.
This releases the target task from the control of the RX850 Pro.
Note that the exd_tsk system call is used when it is necessary for a task to delete itself.

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

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The del_tsk system call is not defined as CF.
E_ID	–35	Invalid ID number specification (maximum number of tasks created < *tskid*)
*E_NOEXS	–52	The target task does not exist.
*E_OBJ	–63	The target task is not in the dormant state.
E_OACV	–66	An unauthorized ID number (*tskid* ≤ 0) was specified.
E_CTX	–69	The del_tsk system call was issued from a non-task.

sta_tsk

Start Task (–23)

Task/non-task

Overview

Activates another task.

C format

```
#include      <stdrx85p.h>
ER            ercd = sta_tsk(ID tskid, INT stacd);
```

Parameters

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

Explanation

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

The target task is scheduled by the RX850 Pro.

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

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

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The sta_tsk system call is not defined as CF.
E_ID	–35	Invalid ID number specification (maximum number of tasks created < *tskid*)
*E_NOEXS	–52	The target task does not exist.
*E_OBJ	–63	The target task is not in the dormant state.
E_OACV	–66	An unauthorized ID number (*tskid* ≤ 0) was specified.

ext_tsk

Exit Task (–21)

Task

Overview

Terminates the task that issued the system call.

C format

```
#include      <stdrx85p.h>
void          ext_tsk();
```

Parameter

None.

Explanation

This system call changes the state of the task from the run state to the dormant state.
The task is excluded from RX850 Pro scheduling.

Remarks 1. This system call initializes the “task creation information” specified at task creation (at configuration or upon the issuance of the cre_tsk system call).

2. If a task is coded in assembly language, perform coding as follows to terminate the issuing task.

```
jr      _ext_tsk
```

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

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

Return value

None.

exd_tsk

Exit and Delete Task (–22)

Task

Overview

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

C format

```
#include      <stdrx85p.h>
void          exd_tsk();
```

Parameter

None.

Explanation

This system call changes the task from the run state to the non-existent state.

This releases the task from the control of the RX850 Pro.

Remark If a task is coded in assembly language, perform coding as follows to terminate or delete the issuing task.

```
jr      _exd_tsk
```

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

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

Return value

None.

ter_tsk

Terminate Task (–25)

Task

Overview

Forcibly terminates another task.

C format

```
#include      <stdrx85p.h>
ER            ercd = ter_tsk(ID tskid);
```

Parameter

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

Explanation

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

Remark This system call initializes the “task creation information” specified at task creation (at configuration or upon the issuance of the cre_tsk system call).

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

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

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The ter_tsk system call is not defined as CF.
E_ID	–35	Invalid ID number specification (maximum number of tasks created < *tskid*)
*E_NOEXS	–52	The target task does not exist.
*E_OBJ	–63	The target task is the task that issued this system call, or the task is in the dormant state.
E_OACV	–66	An unauthorized ID number (*tskid* ≤ 0) was specified.
E_CTX	–69	The ter_tsk system call was issued from a non-task.

dis_dsp

Disable Dispatch (–30)

Task

Overview

Disables dispatch processing.

C format

```
#include      <stdrx85p.h>
ER            ercd = dis_dsp();
```

Parameter

None.

Explanation

This system call disables dispatch processing (task scheduling).
Dispatch processing is disabled until the ena_dsp system call is issued after this system call has been issued.
If a system call such as chg_pri or sig_sem is issued to schedule tasks after the dis_dsp system call is issued but before the ena_dsp system call is issued, the RX850 Pro merely performs operations on a wait queue and delays actual scheduling until the ena_dsp system call is issued, at which time the processing is performed in batch.

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

2. If a system call such as wait_sem and wai_flg is issued, causing the state of the task to change to the wait state after the dis_dsp system call is issued but before the ena_dsp system call is issued, the RX850 Pro returns E_CTX as the return value, regardless of whether the wait conditions are satisfied.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The dis_dsp system call is not defined as CF.
*E_CTX	–69	Context error • The dis_dsp system call was issued from a non-task. • The dis_dsp system call was issued after the loc_cpu system call was issued.

ena_dsp

Enable Dispatch (–29)

Task

Overview

Enables dispatch processing.

C format

```
#include      <stdrx85p.h>
ER            ercd = ena_dsp();
```

Parameter

None.

Explanation

This system call enables dispatch processing (task scheduling).

If a system call such as chg_pri and sig_sem is issued to schedule tasks after the dis_dsp system call is issued but before the ena_dsp system call is issued, the RX850 Pro merely performs operations on a wait queue and delays actual scheduling until the ena_dsp system call is issued, at which time the processing is performed in batch.

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

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The ena_dsp system call is not defined as CF.
*E_CTX	–69	Context error • The ena_dsp system call was issued from a non-task. • The ena_dsp system call was issued after the loc_cpu system call was issued.

chg_pri

Change Priority (–27)

Task/non-task

Overview

Changes the priority of a task.

C format

```
#include      <stdrx85p.h>
ER            ercd = chg_pri(ID tskid, PRI tskpri);
```

Parameters

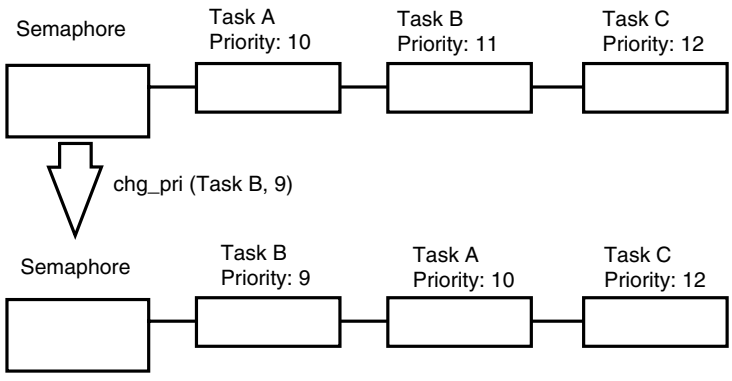
I/O	Parameter	Description
I	ID *tskid;*	Task ID number TSK_SELF(0): Local task Value: Task ID number
I	PRI *tskpri;*	Task priority TPRI_INI(0): Task initial priority Value: Task priority

Explanation

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

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

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



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

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

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The chg_pri system call is not defined as CF.
E_PAR	–33	Invalid priority specification (*tskpri* < 0, maximum priority < *tskpri*)
E_ID	–35	Invalid ID number specification • Maximum number of tasks created < *tskid* • When the chg_pri system call was issued from a non-task, TSK_SELF was specified in *tskid*.
*E_NOEXS	–52	The target task does not exist.
*E_OBJ	–63	The target task is in the dormant state.
E_OACV	–66	An unauthorized ID number (*tskid* < 0) was specified.

rot_rdq

Rotate Ready Queue (–28)

Task/non-task

Overview

Rotates a task ready queue.

C format

```
#include      <stdrx85p.h>
ER            ercd = rot_rdq(PRI tskpri);
```

Parameter

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

Explanation

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

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

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

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The rot_rdq system call is not defined as CF.
E_PAR	–33	Invalid priority specification (*tskpri* < 0, maximum priority < *tskpri*)

rel_wai

Release Wait (–31)

Task/non-task

Overview

Forcibly releases another task from the wait state.

C format

```
#include      <stdrx85p.h>
ER            ercd = rel_wai(ID tskid);
```

Parameter

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

Explanation

This system call forcibly releases the task specified by *tskid* from the wait state.

The target task is excluded from a wait queue, and its state changes from the wait state to the ready state, or from the wait-suspend state to the suspend state.

For a task released from the wait state by the rel_wai system call, E_RLWAI is returned as the return value of the system call (slp_tsk, wai_sem, etc.) that caused transition to the wait state.

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

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The rel_wai system call is not defined as CF.
E_ID	–35	Invalid ID number specification (maximum number of tasks created < *tskid*)
*E_NOEXS	–52	The target task does not exist.
*E_OBJ	–63	The target task is in neither the wait nor wait-suspend state.
E_OACV	–66	An unauthorized ID number (*tskid* ≤ 0) was specified.

get_tid

Get Task Identifier (–24)

Task/non-task

Overview

Acquires a task ID number.

C format

```
#include      <stdrx85p.h>
ER            ercd = get_tid(ID *p_tskid);
```

Parameter

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

Explanation

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

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

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The get_tid system call is not defined as CF.
E_PAR	–33	The address of the area used to store the ID number is invalid (*p_tskid* = 0).

ref_tsk

Refer Task Status (–20)

Task/non-task

Overview

Acquires task information.

C format

```
#include      <stdrx85p.h>
ER            ercd = ref_tsk(T_RTSK *pk_rtsk, ID tskid);
```

Parameters

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

- Structure of task information T_RTSK

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

Explanation

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

The following describes the task information in detail.

exinf ... Extended information

tskpri ... Current priority

tskstat	...	Task state
		TTS_RUN(H'01): Run state
		TTS_RDY(H'02): Ready state
		TTS_WAI(H'04): Wait state
		TTS_SUS(H'08): Suspend state
		TTS_WAS(H'0c): Wait-suspend state
		TTS_DMT(H'10): Dormant state
tskwait	...	Type of wait state
		TTW_SLP(H'0001): Wake-up wait state
		TTW_DLY(H'0002): Timeout wait state
		TTW_FLG(H'0010): Event flag wait state
		TTW_SEM(H'0020): Resource wait state
		TTW_MBX(H'0040): Message wait state
		TTW_MPL(H'1000): Memory block wait state
wid	...	ID number of wait object (semaphore, event, flag, etc.)
wupcnt	...	Number of wake-up requests
suscnt	...	Number of suspend requests
keyid	...	Key ID number
		FALSE(0): No key ID number specified at creation
		Value: Key ID number

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

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

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The ref_tsk system call is not defined as CF.
E_PAR	–33	The start address of the packet used to store task information is invalid (*pk_rtsk* = 0)
E_ID	–35	Invalid ID number specification • Maximum number of tasks created < *tskid* • When the ref_tsk system call was issued from a non-task, TSK_SELF was specified in *tskid*.
*E_NOEXS	–52	The target task does not exist.
E_OACV	–66	An unauthorized ID number (*tskid* < 0) was specified.

vget_tid

Get Task Identifier (–248)

Task/non-task

Overview

Acquires a task ID number.

C format

```
#include       <stdrx85p.h>
ER             ercd = vget_tid(ID *p_tskid, ID keyid);
```

Parameters

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

Explanation

This system call stores the task ID number specified by *keyid* in the area specified by *p_tskid*.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The vget_tid system call is not defined as CF.
E_PAR	–33	Invalid parameter specification • The address of the area used to store the ID number is invalid (*p_tskid* = 0). • Invalid key ID number specification (*keyid* = 0)
*E_NOEXS	–52	The target task does not exist.

11.8.2 Task-associated synchronization system calls

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

Table 11-6 lists the task-associated synchronization system calls.

Table 11-6. Task-Associated Synchronization System Calls

System Call	Function
sus_tsk	Places another task in the suspend state.
rsm_tsk	Restarts a task in the suspend state.
frsm_tsk	Forcibly restarts a task in the suspend state.
slp_tsk	Places the task that issued this system call into the wake-up wait state.
tslp_tsk	Places the task that issued this system call into the wake-up wait state (with timeout).
wup_tsk	Wakes up another task.
can_wup	Invalidates a request to wake up a task.

sus_tsk

Suspend Task (–33)

Task/non-task

Overview

Places another task in the suspend state.

C format

```
#include      <stdrx85p.h>
ER            ercd = sus_tsk(ID tskid);
```

Parameter

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

Explanation

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

If the target task is in the ready or wait state when this system call is issued, this system call changes the target 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 Pro consists of seven bits. Therefore, once the number of suspend requests exceeds 127, the sus_tsk system call returns E_QOVR as the return value without incrementing the suspend request counter.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The sus_tsk system call is not defined as CF.
E_ID	–35	Invalid ID number specification (maximum number of tasks created < *tskid*)
*E_NOEXS	–52	The target task does not exist.
*E_OBJ	–63	Invalid state of the specified task • The target task is in the dormant state. • The issuing task is specified as the target task when the sus_tsk system call is issued from a task.
E_OACV	–66	An unauthorized ID number (*tskid* ≤ 0) was specified.
*E_QOVR	–73	The number of suspend requests exceeded 127.

rsm_tsk

Resume Task (–35)

Task/non-task

Overview

Restarts a task in the suspend state.

C format

```
#include      <stdrx85p.h>
ER            ercd = rsm_tsk(ID tskid);
```

Parameter

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

Explanation

This system call cancels only one of the suspend requests that are issued to the task specified by *tskid* (the suspend request counter is decremented by 0x1).

If the issuance of this system call causes the suspend request counter for the target task to be 0x0, this system call changes the task from the suspend state to the ready state or from the wait-suspend state to the wait state.

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

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The rsm_tsk system call is not defined as CF.
E_ID	–35	Invalid ID number specification (maximum number of tasks created < *tskid*)
*E_NOEXS	–52	The target task does not exist.
*E_OBJ	–63	The target task is not in the suspend or wait-suspend state.
E_OACV	–66	An unauthorized ID number (*tskid* ≤ 0) was specified.

frsm_tsk

Force Resume Task (–36)

Task/non-task

Overview

Forcibly restarts a task in the suspend state.

C format

```
#include       <stdrx85p.h>
ER             ercd = frsm_tsk(ID tskid);
```

Parameter

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

Explanation

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

The target task changes from the suspend state to the read state or from the wait-suspend state to the wait state.

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

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The frsm_tsk system call is not defined as CF.
E_ID	–35	Invalid ID number specification (maximum number of tasks created < *tskid*)
*E_NOEXS	–52	The target task does not exist.
*E_OBJ	–63	The target task is not in the suspend or wait-suspend state.
E_OACV	–66	An unauthorized ID number (*tskid* ≤ 0) was specified.

slp_tsk

Sleep Task (–38)

Task

Overview

Places the task that issued this system call into the wake-up wait state.

C format

```
#include      <stdrx85p.h>
ER            ercd = slp_tsk();
```

Parameter

None.

Explanation

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

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

The wake-up wait state is released 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.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The slp_tsk system call is not defined as CF.
E_CTX	–69	Context error • The slp_tsk system call was issued from a non-task. • The slp_tsk system call was issued in the dispatch disabled state.
*E_RLWAI	–86	The wake-up wait state was forcibly released by the rel_wai system call.

tslp_tsk

Sleep Task with Timeout (–37)

Task

Overview

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

C format

```
#include      <stdrx85p.h>
ER            ercd = tslp_tsk(TMO tmout);
```

Parameter

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

Explanation

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

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

Note that the wake-up wait state is canceled if the wait time specified by tmout elapses or if the wup_tsk, ret_wup, or rel_wai system call is issued, and the issuing task changes to the ready state.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The tslp_tsk system call is not defined as CF.
E_PAR	–33	Invalid wait time specification (*tmout* < TMO_FEVR)
E_CTX	–69	Context error • The tslp_tsk system call was issued from a non-task. • The tslp_tsk system call was issued in the dispatch disabled state.
*E_TMOUT	–85	The wait time has elapsed.
*E_RLWAI	–86	The wake-up wait state was forcibly released by the rel_wai system call.

wup_tsk

Wakeup Task (–39)

Task/non-task

Overview

Wakes up another task.

C format

```
#include      <stdrx85p.h>
ER            ercd = wup_tsk(ID tskid);
```

Parameter

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

Explanation

This system call issues a wake-up request to the task specified by *tskid* (the wake-up request counter is incremented by 0x1).

If the target 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 (incrementing the wake-up request counter).

Caution The wake-up request counter managed by the RX850 Pro consists of 7-bits. Therefore, when the number of wake-up requests exceeds 127, the wup_tsk system call returns E_QOVR as the return value without incrementing the wake-up request counter.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The wup_tsk system call is not defined as CF.
E_ID	–35	Invalid ID number specification (maximum number of tasks created < *tskid*)
*E_NOEXS	–52	The target task does not exist.
*E_OBJ	–63	Invalid state of the specified task • The target task is in the dormant state. • The issuing task is specified as the target task when the wup_tsk system call is issued from a task.
E_OACV	–66	An unauthorized ID number (*tskid* ≤ 0) was specified.
*E_QOVR	–73	The number of wake-up requests exceeded 127.

can_wup

Cancel Wakeup Task (–40)

Task/non-task

Overview

Invalidates a request to wake up a task.

C format

```
#include      <stdrx85p.h>
ER            ercd = 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 TSK_SELF(0): Local task Value: Task ID number

Explanation

This system call cancels all the wake-up requests issued to the task specified by *tskid* (the wake-up request counter is set to 0x0).

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

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The can_wup system call is not defined as CF.
E_PAR	–33	The address of the area used to store the number of wake-up requests is invalid (*p_wupcnt* = 0).
E_ID	–35	Invalid ID number specification • Maximum number of tasks created < *tskid* • When the can_wup system call was issued from a non-task, TSK_SELF was specified for *tskid*.
*E_NOEXS	–52	The target task does not exist.
*E_OBJ	–63	The target task is in the dormant state.
E_OACV	–66	An unauthorized ID number (*tskid* < 0) was specified.

11.8.3 Synchronous communication system calls

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

Table 11-7 lists the synchronous communication system calls.

Table 11-7. Synchronous Communication System Calls

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

cre_sem

Create Semaphore (–49)

Task

Overview

Creates a semaphore.

C format

- When an ID number is specified

```
#include      <stdrx85p.h>
ER            ercd = cre_sem(ID semid, T_CSEM *pk_csem);
```

- When an ID number is not specified

```
#include      <stdrx85p.h>
ER            ercd = cre_sem(ID_AUTO, T_CSEM *pk_csem, ID *p_semid);
```

Parameters

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

- Structure of semaphore creation information T_CSEM

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

Explanation

The RX850 Pro provides two types of interfaces for semaphore creation: one in which an ID number must be specified, and one in which an ID number is not specified.

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

- When an ID number is not specified
  A semaphore is created based on the information specified by *pk_csem*.
  An ID number is allocated by the RX850 Pro and the allocated ID number is stored in the area specified by *p_semid*.

Semaphore creation information is described in detail below.

exinf ... Extended information

An area for storing user-specific information on a target semaphore. The user can use this area as required.

Information set in exinf can be dynamically acquired by issuing the ref_sem system call from a processing program (tasks and non-tasks).

sematr ... Semaphore attribute

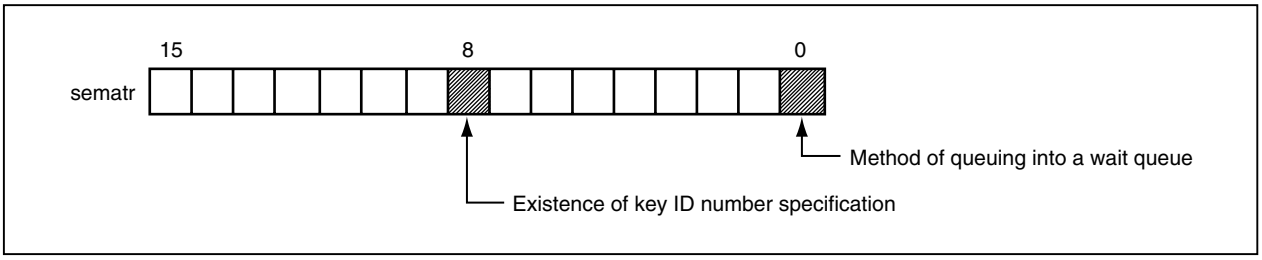
Bit 0 .. Method of queuing into a wait queue

TA_TPRI(0): Priority order

TA_TFIFO(1): FIFO order

Bit 8 .. Existence of specifying the key ID number

TA_KEYID(1): Specifies the key ID number



isemcnt ... Initial semaphore resource count

maxsem ... Maximum semaphore resource count

keyid ... Semaphore key ID number

Remark If the value of bit 8 is not TA_KEYID in sematr, the contents of keyid are meaningless.

Return value

*E_OK	0	Normal termination
*E_NOMEM	–10	The semaphore management block area cannot be secured.
*E_NOSPT	–17	The cre_sem system call is not defined as CF.
E_RSATR	–24	Invalid specification of attribute sematr
E_PAR	–33	Invalid parameter specification • The start address of a packet storing semaphore creation information is invalid (*pk_csem* = 0). • The initial resource count is invalid (isemcnt < 0). • The maximum resource count is invalid (maxsem ≤ 0, maxsem < isemcnt). • Invalid key ID number specification (keyid = 0) (when TA_KEYID attribute specified) • The address of the area used to store an ID number is invalid (*p_semid* = 0). (When a semaphore is created without an ID number specified)
E_ID	–35	Invalid ID number specification (maximum number of semaphores created < *semid*)
*E_OBJ	–63	A semaphore having the specified ID number has already been created.
E_OACV	–66	An unauthorized ID number (*semid* ≤ 0) was specified.
E_CTX	–69	The cre_sem system call was issued from a non-task.

del_sem

Delete Semaphore (–50)

Task

Overview

Deletes a semaphore.

C format

```
#include      <stdrx85p.h>
ER            ercd = del_sem(ID semid);
```

Parameter

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

Explanation

This system call deletes the semaphore specified by *semid*.

The target semaphore is released from the control of the RX850 Pro.

The task released from the wait state (resource wait state) by the del_sem system call has E_DLT returned as the return value of the system call (wai_sem or twai_sem) that initiated transition to the wait state.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The del_sem system call is not defined as CF.
E_ID	–35	Invalid ID number specification (maximum number of semaphores created < *semid*)
*E_NOEXS	–52	The target semaphore does not exist.
E_OACV	–66	An unauthorized ID number (*semid* ≤ 0) was specified.
E_CTX	–69	The del_sem system call was issued from a non-task.

sig_sem

Signal Semaphore (–55)

Task/non-task

Overview

Returns resources.

C format

```
#include      <stdrx85p.h>
ER            ercd = sig_sem(ID semid);
```

Parameter

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

Explanation

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

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

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

Caution The semaphore counter managed by the RX850 Pro counts up to the maximum number of resources that can be acquired as specified at the time it is created. Therefore, when the number of resources exceeds the maximum number of resources, by issuing the sig_sem system call, E_QOVR is returned as the return value without incrementing the semaphore counter.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The sig_sem system call is not defined as CF.
E_ID	–35	Invalid ID number specification (maximum number of semaphores created < *semid*)
*E_NOEXS	–52	The target semaphore does not exist.
E_OACV	–66	An unauthorized ID number (*semid* ≤ 0) was specified.
*E_QOVR	–73	The resource count exceeded the maximum resource count specified at creation.

wai_sem

Wait on Semaphore (–53)

Task

Overview

Acquires resources.

C format

```
#include      <stdrx85p.h>
ER            ercd = wai_sem(ID semid);
```

Parameter

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

Explanation

This system call acquires resources from the semaphore specified by *semid* (the semaphore counter is decremented by 0x1).

When this system call is issued, if no resource can be acquired from a target semaphore (when there are no free resources), this system call places the task in the wait 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 issuance of the sig_sem, del_sem, or rel_wai system call, and the task returns to the ready state.

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

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The wai_sem system call is not defined as CF.
E_ID	–35	Invalid ID number specification (maximum number of semaphores created < *semid*)
*E_NOEXS	–52	The target semaphore does not exist.
E_OACV	–66	An unauthorized ID number (*semid* ≤ 0) was specified.
E_CTX	–69	Context error • The wai_sem system call was issued from a non-task. • The wai_sem system call was issued in the dispatch disabled state.
*E_DLT	–81	The specified semaphore was deleted by the del_sem system call.
*E_RLWAI	–86	The resource wait state was forcibly released by the rel_wai system call.

preq_sem

Poll and Request Semaphore (–107)

Task/non-task

Overview

Acquires resources (polling).

C format

```
#include      <stdrx85p.h>
ER            ercd = preq_sem(ID semid);
```

Parameter

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

Explanation

This system call acquires resources from the semaphore specified by *semid* (the semaphore counter is decremented by 0x1).

When this system call is issued, if no resource can be acquired from a target semaphore (when there are no free resources), this system call returns E_TMOUT as the return value.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The preq_sem system call is not defined as CF.
E_ID	–35	Invalid ID number specification (maximum number of semaphores created < *semid*)
*E_NOEXS	–52	The target semaphore does not exist.
E_OACV	–66	An unauthorized ID number (*semid* ≤ 0) was specified.
*E_TMOUT	–85	The resource count for the target semaphore is 0x0.

twai_sem

Wait on Semaphore with Timeout (–171)

Task

Overview

Acquires resources (with timeout).

C format

```
#include      <stdrx85p.h>
ER            ercd = twai_sem(ID semid, TMO tmout);
```

Parameters

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

Explanation

This system call acquires resources from the semaphore specified by *semid* (the semaphore counter is decremented by 0x1).

When this system call is issued, if no resource can be acquired from a target semaphore (when there are no free resources), this system call places the task in the wait queue of the target 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 by *tmout* elapses or when the sig_sem, del_sem, or rel_wai system call is issued, at which time it changes to the ready state.

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

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The twai_sem system call is not defined as CF.
E_PAR	–33	Invalid wait time specification (*tmout* < TMO_FEVR)
E_ID	–35	Invalid ID number specification (maximum number of semaphores created < *semid*)
*E_NOEXS	–52	The target semaphore does not exist.
E_OACV	–66	An unauthorized ID number (*semid* ≤ 0) was specified.
E_CTX	–69	Context error • The twai_sem system call was issued from a non-task. • The twai_sem system call was issued in the dispatch disabled state.
*E_DLT	–81	A target semaphore was deleted by the del_sem system call.
*E_TMOUT	–85	Wait time elapsed.
*E_RLWAI	–86	The resource wait state was forcibly released by the rel_wai system call.

ref_sem

Refer Semaphore Status (–52)

Task/non-task

Overview

Acquires semaphore information.

C format

```
#include      <stdrx85p.h>
ER            ercd = ref_sem(T_RSEM *pk_rsem, ID semid);
```

Parameters

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

- Structure of semaphore information T_RSEM

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

Explanation

This system call stores the semaphore information (extended information, existence of waiting task, etc.) for the semaphore specified by *semid* in the packet specified by *pk_rsem*.
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 wait queue
semcnt ... Current resource count
maxsem ... Maximum resource count specified at creation
keyid ... Key ID number
  FALSE(0): No key ID number specified at creation
  Value: Key ID number

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The ref_sem system call is not defined as CF.
E_PAR	–33	The start address of the packet used to store semaphore information is invalid (*pk_rsem* = 0).
E_ID	–35	Invalid ID number specification (maximum number of semaphores created < *semid*)
*E_NOEXS	–52	The target semaphore does not exist.
E_OACV	–66	An unauthorized ID number (*semid* ≤ 0) was specified.

vget_sid

Get Semaphore Identifier (–246)

Task/non-task

Overview

Acquires the semaphore ID number.

C format

```
#include      <stdrx85p.h>
ER            ercd = vget_sid(ID *p_semid, ID keyid);
```

Parameters

I/O	Parameter		Description
O	ID	*p_semid;	Address of an area used to store an ID number
I	ID	keyid;	Semaphore key ID number

Explanation

This system call stores the semaphore ID number specified by *keyid* in the area specified by *p_semid*.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The vget_sid system call is not defined as CF.
E_PAR	–33	The start address of the packet used to store semaphore information is invalid (*pk_rsem* = 0). • Invalid key ID number specification (*keyid* = 0) • The address of the area used to store the ID number is invalid (*p_semid* = 0).
*E_NOEXS	–52	The target semaphore does not exist.

cre_flg

Create Event Flag (–41)

Task

Overview

Creates an event flag.

C format

- When an ID number is specified

```
#include        <stdrx85p.h>
ER              ercd = cre_flg(ID flgid, T_CFLG *pk_cflg);
```

- When an ID number is not specified

```
#include        <stdrx85p.h>
ER              ercd = cre_flg(ID_AUTO, T_CFLG *pk_cflg, ID *p_flgid);
```

Parameters

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

- Structure of event flag creation information T_CFLG

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

Explanation

The RX850 Pro provides two types of interfaces for event flag creation: one in which an ID number must be specified and one in which an ID number is not specified.

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

- When an ID number is not specified
  An event flag is created based on the information specified by *pk_cflg*.
  An ID number is allocated by the RX850 Pro and the allocated ID number is stored in the area specified by *p_flgid*.

Event flag creation information is described in detail below.

exinf ... Extended information

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

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

flgatr ... Event flag attribute

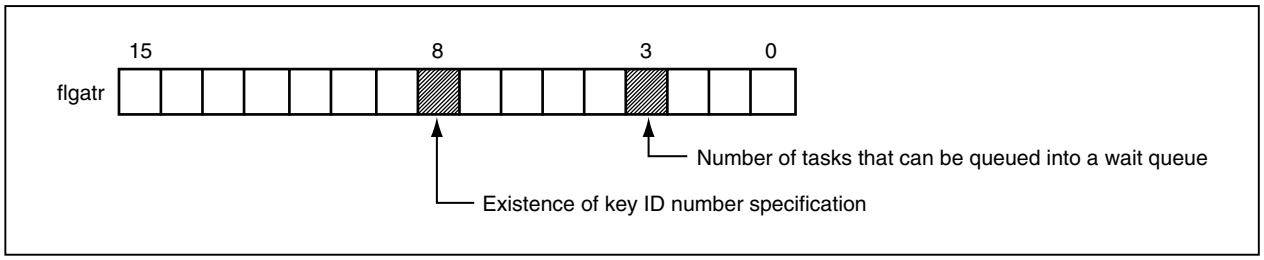
Bit 3 .. Number of tasks that can be queued into a wait queue

TA_WSGL(0): One task only

TA_WMUL(1): Two or more tasks

Bit 8 .. Existence of key ID number specification

TA_KEYID(1): Key ID number specified



iflgptn ... Initial bit pattern of event flag

keyid ... Event flag key ID number

Remark If the value of bit 8 is not TA_KEYID in flgatr, the contents of keyid are meaningless.

Return value

*E_OK	0	Normal termination
*E_NOMEM	–10	The event flag management block area cannot be secured.
*E_NOSPT	–17	The cre_flg system call is not defined as CF.
E_RSATR	–24	Invalid specification of attribute flgatr
E_PAR	–33	Invalid parameter specification • The start address of the packet storing event flag creation information is invalid (*pk_cflg* = 0). • Invalid key ID number specification (keyid = 0) (when TA_KEYID attribute specified) • The address of the area used to store the ID number is invalid (*p_flgid* = 0). (When an event flag is created with no ID number specified)
E_ID	–35	Invalid ID number specification (maximum number of event flags created < *flgid*)
*E_OBJ	–63	An event flag having the specified ID number has already been created.
E_OACV	–66	An unauthorized ID number (*flgid* ≤ 0) was specified.
E_CTX	–69	The cre_flg system call was issued from a non-task.

del_flg

Delete Event Flag (–42)

Task

Overview

Deletes an event flag.

C format

```
#include      <stdrx85p.h>
ER            ercd = del_flg(ID flgid);
```

Parameter

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

Explanation

This system call deletes the event flag specified by *flgid*.

The target event flag is released from the control of the RX850 Pro.

The task released from the wait state (event flag wait state) by this system call has E_DLT returned as the return value of the system call (wai_flg or twai_flg) that initiated transition to the wait state.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The del_flg system call is not defined as CF.
E_ID	–35	Invalid ID number specification (maximum number of event flags created < *flgid*)
*E_NOEXS	–52	The target event flag does not exist.
E_OACV	–66	An unauthorized ID number (*flgid* ≤ 0) was specified.
E_CTX	–69	The del_flg system call was issued from a non-task.

set_flg

Set Event Flag (–48)

Task/non-task

Overview

Sets a bit pattern.

C format

```
#include      <stdrx85p.h>
ER            ercd = set_flg(ID flgid, UINT setptn);
```

Parameters

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

Explanation

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

For example, when this system call is issued, if the target event flag's bit pattern is B'1100 and the bit pattern specified by *setptn* is B'1010, the bit pattern of the target event flag becomes B'1110.

When this system call is issued, if the wait condition for a task queued in the wait queue of the target event flag is satisfied, the task is removed from the wait queue.

Consequently, the relevant task changes from the wait state (event flag wait state) to the ready state, or from the wait-suspend state to the suspend state.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The set_flg system call is not defined as CF.
E_ID	–35	Invalid ID number specification (maximum number of event flags created < *flgid*)
*E_NOEXS	–52	The target event flag does not exist.
E_OACV	–66	An unauthorized ID number (*flgid* ≤ 0) was specified.

clr_flg

Clear Event Flag (–47)

Task/non-task

Overview

Clears a bit pattern.

C format

```
#include      <stdrx85p.h>
ER            ercd = clr_flg(ID flgid, UINT clrptn);
```

Parameters

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

Explanation

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

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

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The clr_flg system call is not defined as CF.
E_ID	–35	Invalid ID number specification (maximum number of event flags created < *flgid*)
*E_NOEXS	–52	The target event flag does not exist.
E_OACV	–66	An unauthorized ID number (*flgid* ≤ 0) was specified.

wai_flg

Wait Event Flag (–46)

Task

Overview

Checks a bit pattern.

C format

```
#include      <stdrx85p.h>
ER            ercd =  wai_flg(UINT *p_flgptn, ID flgid, UINT waiptn, UINT wfmode);
```

Parameters

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

Explanation

This system call checks whether a bit pattern that satisfies the request bit pattern specified by *waiptn*, as well as the wait condition specified by *wfmode*, is set in the event flag specified by *flgid*.

If a bit pattern satisfying the wait condition is set in the target event flag, this system call stores the bit pattern of the event flag in the area specified by *p_flgptn*.

When this system call is issued, if the bit pattern of the target event flag does not satisfy the wait condition, this system call queues the task at the end of the wait queue for the target event flag, then changes it from the run state to the wait state (event flag wait state).

The event flag wait state is released when a bit pattern satisfying the wait condition is set by the set_flg system call, or when the del_flg or rel_wai system call is issued, at which time it changes to the ready state.

The specification format for *wfmode* is shown below.

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

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

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

- *wfmode* = (TWF_ORW|TWF_CLR)
  This system call checks whether at least one of the bits of *waiptn* that are set to 1 is set in the target event flag. If the wait condition is satisfied, the bit pattern of the target event flag is cleared (B'0000 is set).

Cautions 1. The RX850 Pro specifies the number of tasks that can be queued into the wait queue of an event flag at creation (at configuration or upon the issuance of the cre_flg system call).

TA_WSGL attribute: Only one task can be queued.
TA_WMUL attribute: Two or more tasks can be queued.

For this reason, if this system call is issued for the event flag having the TA_WSGL attribute for which waiting tasks are already queued, the wai_flg system call returns E_OBJ as the return value without performing bit pattern checking.

2. If the event flag wait state is forcibly released by issuing the del_flg or rel_wai system call, the contents of the area specified by *p_flgptn* will be undefined.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The wai_flg system call is not defined as CF.
E_PAR	–33	Invalid parameter specification • The address of the area used to store a bit pattern when a condition is satisfied is invalid (*p_flgptn* = 0). • Invalid specification of request bit pattern (*waiptn* = 0). • Invalid specification of wait condition or condition satisfaction parameter *wfmode*.
E_ID	–35	Invalid ID number specification (maximum number of event flags created < *flgid*)
*E_NOEXS	–52	The target event flag does not exist.
*E_OBJ	–63	The wai_flg system call was issued for the event flag having the TA_WSGL attribute for which waiting tasks were already queued.
E_OACV	–66	An unauthorized ID number (*flgid* ≤ 0) was specified.
E_CTX	–69	Context error • The wai_flg system call was issued from a non-task. • The wai_flg system call was issued from the dispatch disabled state.
*E_DLT	–81	The target event flag was deleted by the del_flg system call.
*E_RLWAI	–86	The event flag wait state was forcibly released by the rel_wai system call.

pol_flg

Poll Event Flag (–106)

Task/non-task

Overview

Checks a bit pattern (polling).

C format

```
#include      <stdrx85p.h>
ER            ercd =  pol_flg(UINT *p_flgptn, ID flgid, UINT waiptn, UINT wfmode);
```

Parameters

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

Explanation

This system call checks whether a bit pattern satisfying both the request bit pattern specified by *waiptn* and the wait condition specified by *wfmode* is set in the event flag specified by *flgid*.

If a bit pattern satisfying the wait condition is set in the target event flag, this system call stores the bit pattern of the event flag into the area specified by *p_flgptn*.

When this system call is issued, if the bit pattern of the target 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 the bits of *waiptn* that are set to 1 are set in the target event flag.

- *wfmode* = (TWF_ANDW|TWF_CLR)
  This system call checks whether all the bits of *waiptn* that are set to 1 are set in the target 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 the bits of *waiptn* that are set to 1 is set in the target event flag.

- *wfmode* = (TWF_ORW|TWF_CLR)
  This system call checks whether at least one of the bits of *waiptn* that are set to 1 is set in the target event flag. If the wait condition is satisfied, the bit pattern for the target event flag is cleared (B'0000 is set).

Caution The RX850 Pro specifies the number of tasks that can be queued into the wait queue of an event flag at creation (at configuration or upon the issuance of the cre_flg system call).

TA_WSGL attribute: Only one task can be queued.
TA_WMUL attribute: Two or more tasks can be queued.

For this reason, if this system call is issued for an event flag having the TA_WSGL attribute for which waiting tasks are already queued, the wai_flg system call returns E_OBJ as the return value without performing bit pattern checking.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The pol_flg system call is not defined as CF.
E_PAR	–33	Invalid parameter specification • The address of the area used to store a bit pattern when a condition is satisfied is invalid (*p_flgptn* = 0). • Invalid specification of request bit pattern (*waiptn* = 0). • Invalid specification of wait condition or condition satisfaction parameter *wfmode*.
E_ID	–35	Invalid ID number specification (maximum number of event flags created < *flgid*)
*E_NOEXS	–52	The target event flag does not exist.
*E_OBJ	–63	This pol_flg system call was issued for the event flag of TA_WSGL attribute for which waiting tasks are already queued.
E_OACV	–66	An unauthorized ID number (*flgid* ≤ 0) was specified.
*E_TMOUT	–85	The bit pattern of the target event flag does not satisfy the wait condition.

twai_flg

Wait Event Flag with Timeout (–170)

Task

Overview

Checks a bit pattern (with timeout).

C format

```
#include      <stdrx85p.h>
ER            ercd = twai_flg(UINT *p_flgptn, ID flgid, UINT waiptn, UINT wfmode,
              TMO tmout);
```

Parameters

I/O	Parameter		Description
O	UINT	*p_flgptn;	Address of area used to store bit pattern when condition is satisfied
I	ID	flgid;	Event flag ID number
I	UINT	waiptn;	Request bit pattern (32-bit width)
I	UINT	wfmode;	Wait condition or condition satisfaction TWF_ANDW(0): AND wait TWF_ORW(2): OR wait TWF_CLR(1): Bit pattern is cleared.
I	TMO	tmout;	Wait time (unit: ms) 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 by *waiptn* and the wait condition specified by *wfmode* is set in the event flag specified by *flgid*.

If a bit pattern satisfying the wait condition is set in the target event flag, this system call stores the bit pattern of the event flag in the area specified by *p_flgptn*.

Upon the issuance of this system call, if the bit pattern of the target event flag does not satisfy the wait condition, this system call queues the task at the end of the wait queue for the target 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 by *tmout*, when a bit pattern satisfying the wait condition is set by the set_flg system call, or when the del_flg or rel_wai system call is issued, at which time the task returns to the ready state.

The *wfmode* specification format is shown below.

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

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

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

- *wfmode* = (TWF_ORW|TWF_CLR)
  This system call checks whether at least one of the bits of *waiptn* that are set to 1 is set in the target event flag.
  If the wait condition is satisfied, the bit pattern of the target event flag is cleared (B'0000 is set).

Cautions 1. The RX850 Pro specifies the number of tasks that can be queued into the wait queue of the event flag at creation (at configuration or upon the issuance of the cre_flg system call).

TA_WSGL attribute: Only one task can be queued.
TA_WMUL attribute: Two or more tasks can be queued.

For this reason, if this system call is issued for an event flag having the TA_WSGL attribute for which waiting tasks are already queued, this system call returns E_OBJ as the return value without performing bit pattern checking.

2. If the event flag wait state is forcibly released by the del_flg or rel_wai system call, the contents of the area specified by *p_flgptn* will be undefined.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The twai_flg system call is not defined as CF.
E_PAR	–33	Invalid parameter specification • The address of the area used to store the bit pattern when the condition is satisfied is invalid (*p_flgptn* = 0). • The specification of the request bit pattern is invalid (*waiptn* = 0). • The specification of the wait condition or condition satisfaction parameter *wfmode* is invalid. • Invalid wait time specification (*tmout* < TMO_FEVR)
E_ID	–35	Invalid ID number specification (maximum number of event flags created < *flgid*)
*E_NOEXS	–52	The target event flag does not exist.
*E_OBJ	–63	This twai_flg system call was issued for the event flag having the TA_WSGL attribute in which waiting tasks were already queued.
E_OACV	–66	An unauthorized ID number (*flgid* ≤ 0) was specified.
E_CTX	–69	Context error • The twai_flg system call was issued from a non-task. • The twai_flg system call was issued from the dispatch disabled state.
*E_DLT	–81	The specified event flag was deleted by the del_flg system call.
*E_TMOUT	–85	Wait time elapsed.
*E_RLWAI	–86	The event flag wait state was forcibly released by the rel_wai system call.

ref_flg

Refer Event Flag Status (–44)

Task/non-task

Overview

Acquires event flag information.

C format

```
#include      <stdrx85p.h>
ER            ercd = ref_flg(T_RFLG *pk_rflg, ID flgid);
```

Parameters

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

- Structure of event flag information T_RFLG

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

Explanation

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

flgptn ... Current bit pattern

keyid ... Key ID number

- FALSE(0): No key ID number specified at generation
- Value: Key ID number

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The ref_flg system call is not defined as CF.
E_PAR	–33	The start address of the packet used to store event flag information is invalid (*pk_rflg* = 0).
E_ID	–35	Invalid ID number specification (maximum number of event flags created < *flgid*)
*E_NOEXS	–52	The target event flag does not exist.
E_OACV	–66	An unauthorized ID number (*flgid* ≤ 0) was specified.

vget_fid

Get Event Flag Identifier (–247)

Task/non-task

Overview

Acquires the event flag ID number.

C format

```
#include      <stdrx85p.h>
ER            ercd = vget_fid(ID *p_flgid, ID keyid);
```

Parameters

I/O	Parameter		Description
O	ID	**p_flgid;*	Address of area used to store ID number
I	ID	*keyid;*	Event flag key ID number

Explanation

This system call stores the event flag ID number specified by *keyid* in the area specified by *p_flgid*.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The vget_fid system call is not defined as CF.
E_PAR	–33	Invalid parameter specification • Invalid key ID number specification (*keyid* = 0) • The address of the area used to store the ID number is invalid (*p_flgid* = 0).
*E_NOEXS	–52	The target event flag does not exist.

cre_mbx

Create Mailbox (–57)

Task

Overview

Creates a mailbox.

C format

- When an ID number is specified

```
#include        <stdrx85p.h>
ER              ercd = cre_mbx(ID mbxid, T_CMBX *pk_cmbx);
```

- When an ID number is not specified

```
#include              <stdrx85p.h>
ER                    ercd = cre_mbx(ID_AUTO, T_CMBX *pk_cmbx, ID *p_mbxid);
```

Parameters

I/O	Parameter		Description
I	ID	*mbxid;*	Mailbox ID number
I	T_CMBX	**pk_cmbx;*	Start address of packet used to store mailbox creation information
O	ID	**p_mbxid;*	Address of area used to store ID number

- Structure of mailbox creation information T_CMBX

```
typedef    struct     t_cmbx {
           VP         exinf;      /*  Extended information        */
           ATR        mbxatr;     /*  Mailbox attribute           */
           ID         keyid;      /*  Mailbox key ID number       */
} T_CMBX;
```

Explanation

The RX850 Pro provides two types of interfaces for mailbox creation: one in which an ID number must be specified for mailbox creation, and one in which an ID number is not specified.

- When an ID number is specified
  A mailbox having the ID number specified by *mbxid* is created based on the information specified by *pk_cmbx*.

- When an ID number is not specified
  A mailbox is created based on the information specified by *pk_cmbx*.
  An ID number is allocated by the RX850 Pro. The allocated ID number is stored in the area specified by *p_mbxid*.

Mailbox creation information is described in detail below.

exinf ... Extended information

exinf is an area used for storing user-specific information on the target mailbox. The user can use this area as required.

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

mbxatr ... Mailbox attribute

Bit 0 .. Method of queuing into a task wait queue

TA_TPRI(0): Priority order

TA_TFIFO(1): FIFO order

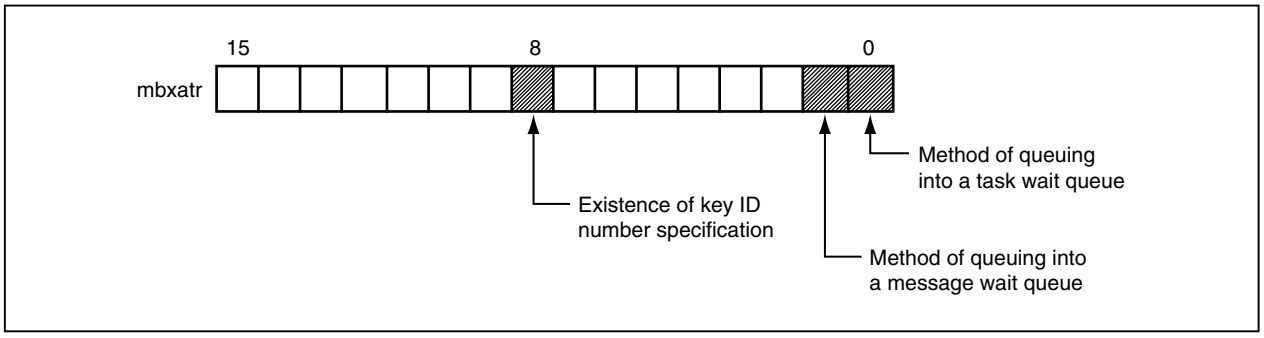
Bit 1 .. Method of queuing into a message wait queue

TA_MPRI(0): Priority order

TA_MFIFO(1): FIFO order

Bit 8 .. Existence of key ID number specification

TA_KEYID(1): Key ID number specified



keyid ... Mailbox key ID number

Remark If the value of bit 8 is not TA_KEYID in mbxatr, the contents of keyid are meaningless.

Return value

*E_OK	0	Normal termination
*E_NOMEM	–10	The mailbox management block area cannot be secured.
*E_NOSPT	–17	The cre_mbx system call is not defined as CF.
E_RSATR	–24	Invalid specification of attribute mbxatr
E_PAR	–33	Invalid parameter specification • The start address of the packet storing the mailbox creation information is invalid (*pk_cmbx* = 0). • The specification of the key ID number is invalid (keyid = 0) (when TA_KEYID specified). • The address of the area used to store the ID number is invalid (*p_mbxid* = 0). (When a mailbox is created without an ID number specified).
E_ID	–35	Invalid ID number specification (maximum number of mailboxes created < *mbxid*)
*E_OBJ	–63	A mailbox having the specified ID number has already been created.
E_OACV	–66	An unauthorized ID number (*mbxid* ≤ 0) was specified.
E_CTX	–69	The cre_mbx system call was issued from a non-task.

del_mbx

Delete Mailbox (–58)

Task

Overview

Deletes a mailbox.

C format

```
#include      <stdrx85p.h>
ER            ercd = del_mbx(ID mbxid);
```

Parameter

I/O	Parameter	Description
I	ID mbxid;	Mailbox ID number

Explanation

This system call deletes the mailbox specified by *mbxid*.
The target mailbox is released from the control of the RX850 Pro.
The task released from the wait state (message wait state) by this system call has E_DLT returned as the return value of the system call (rcv_msg or trcv_msg) that instigated the transition to the wait state.

Remark When this system call is issued, any message using a memory block acquired from a memory pool is queued into the message wait queue of the target mailbox, and the message (memory block) is then returned to the memory pool.
For this reason, if this system call uses an area other than memory blocks acquired from the memory pool, operation is not guaranteed. This system call should therefore not be issued in the above case.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The del_mbx system call is not defined as CF.
E_ID	–35	Invalid ID number specification (maximum number of mailboxes created < *mbxid*)
*E_NOEXS	–52	The target mailbox does not exist.
E_OACV	–66	An unauthorized ID number (*mbxid* ≤ 0) was specified.
E_CTX	–69	The del_mbx system call was issued from a non-task.

snd_msg

Send Message (–63)

Task/non-task

Overview

Transmits a message.

C format

```
#include      <stdrx85p.h>
ER            ercd = snd_msg(ID mbxid, T_MSG *pk_msg);
```

Parameters

I/O	Parameter		Description
I	ID	*mbxid;*	Mailbox ID number
I	T_MSG	**pk_msg;*	Start address of packet used to store a message

- Structure of message T_MSG

```
typedef    struct     t_msg {
           VW         msgrfu;      /*  Message management area    */
           PRI        msgpri;      /*  Message priority           */
           VB         msgcont[];   /*  Message body               */
} T_MSG;
```

Explanation

This system call transmits the message specified in *pk_msg* to the mailbox specified in *mbxid* (queues the message into a message wait queue).

When this system call is issued, if a task is queued into the task wait queue of the target mailbox, this system call passes the message to the task (first task in the task wait queue) without performing message queuing.

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

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

Caution The RX850 Pro uses the first four bytes (message management area msgrfu) of a message as a link area for enabling queuing into a message wait queue. Accordingly, transmitting a message to the target mailbox requires that 0x0 be set in msgrfu before issuing the snd_msg system call. If a value other than 0x0 is set in msgrfu when the snd_msg system call is issued, the RX850 Pro recognizes that the relevant message is already queued into a message wait queue, and this system call returns E_OBJ as the return value without transmitting the message.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The snd_msg system call is not defined as CF.
E_PAR	–33	The start address of the packet used to store the message is invalid (*pk_msg* = 0).
E_ID	–35	Invalid ID number specification (maximum number of mailboxes created < *mbxid*)
*E_NOEXS	–52	The target mailbox does not exist.
E_OBJ	–63	The area specified for a message is already being used for messages.
E_OACV	–66	An unauthorized ID number (*mbxid* ≤ 0) was specified.

rcv_msg

Receive Message from Mailbox (–61)

Task

Overview

Receives a message.

C format

```
#include      <stdrx85p.h>
ER            ercd = rcv_msg(T_MSG **ppk_msg, ID mbxid);
```

Parameters

I/O	Parameter		Description
O	T_MSG	**ppk_msg;	Address of area used to store start address of message
I	ID	mbxid;	Mailbox ID number

Explanation

This system call receives a message from the mailbox specified by *mbxid* and stores its start address in the area specified by *ppk_msg*.

When this system call is issued, if a message cannot be received from the target mailbox (when no message exists in a message wait queue), this system call queues the task into the task wait queue of the target mailbox, then changes its state from the run state to the wait state (message wait state).

The message wait state is released when the snd_msg, del_mbx, or rel_wai system call is issued, and the task returns to the ready state.

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

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The rcv_msg system call is not defined as CF.
E_PAR	–33	The address of the area used to store the start address of a message is invalid (*ppk_msg* = 0).
E_ID	–35	Invalid ID number specification (maximum number of mailboxes created < *mbxid*)
*E_NOEXS	–52	The target mailbox does not exist.
E_OACV	–66	An unauthorized ID number (*mbxid* ≤ 0) was specified.
E_CTX	–69	Context error • The rcv_msg system call was issued from a non-task. • The rcv_msg system call was issued from the dispatch disabled state.
*E_DLT	–81	The target mailbox was deleted by a del_mbx system call.
*E_RLWAI	–86	The message wait state was forcibly released by the rel_wai system call.

Poll and Receive Message from Mailbox (–108)

prcv_msg

Task/non-task

Overview

Receives a message (polling).

C format

```
#include      <stdrx85p.h>
ER            ercd = prcv_msg(T_MSG **ppk_msg, ID mbxid);
```

Parameters

I/O	Parameter		Description
O	T_MSG	**ppk_msg;	Address of area used to store the start address of a message
I	ID	mbxid;	Mailbox ID number

Explanation

This system call receives a message from the mailbox specified by *mbxid* and stores its start address in the area specified by *ppk_msg*.

When this system call is issued, if a message cannot be received from the target mailbox (when no message exists in the message wait queue), E_TMOUT is returned as the return value.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The prcv_msg system call is not defined as CF.
E_PAR	–33	The address of the area used to store the start address of the message is invalid (*ppk_msg* = 0).
E_ID	–35	Invalid ID number specification (maximum number of mailboxes created < *mbxid*)
*E_NOEXS	–52	A target mailbox does not exist.
E_OACV	–66	An unauthorized ID number (*mbxid* ≤ 0) was specified.
*E_TMOUT	–85	No message exists in the target mailbox.

trcv_msg

Receive Message from Mailbox with Timeout (–172)

Task

Overview

Receives a message (with timeout).

C format

```
#include      <stdrx85p.h>
ER            ercd = trcv_msg(T_MSG **ppk_msg, ID mbxid, TMO tmout);
```

Parameters

I/O	Parameter		Description
O	T_MSG	**ppk_msg;	Address of area used to store start address of message
I	ID	mbxid;	Mailbox ID number
I	TMO	tmout;	Wait time (unit: basic clock cycles) TMO_POL(0): Quick return TMO_FEVR(–1): Permanent wait Value: Wait time

Explanation

This system call receives a message from the mailbox specified by *mbxid* and stores its start address in the area specified by *ppk_msg*.

When this system call is issued, if a message cannot be received from the target mailbox (when no message exists in the message wait queue), this system call queues the task into the task wait queue of the target 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 by *tmout* elapses or when the snd_msg, del_mbx, or rel_wai system call is issued, and the task returns to the ready state.

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

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The trcv_msg system call is not defined as CF.
E_PAR	–33	Invalid parameter specification • The address of the area used to store the start address of a message is invalid (*ppk_msg* = 0).
E_ID	–35	Invalid ID number specification (maximum number of mailboxes created < *mbxid*)
*E_NOEXS	–52	The target mailbox does not exist.
E_OACV	–66	An unauthorized ID number (*mbxid* ≤ 0) was specified.
E_CTX	–69	Context error • The trcv_msg system call was issued from a non-task. • The trcv_msg system call was issued from the dispatch disabled state.
*E_DLT	–81	The specified mailbox was deleted by the del_mbx system call.
*E_TMOUT	–85	The wait time has elapsed.
*E_RLWAI	–86	The message wait state was forcibly released by the rel_wai system call.

ref_mbx

Refer Mailbox Status (–60)

Task/non-task

Overview

Acquires mailbox information.

C format

```
#include      <stdrx85p.h>
ER            ercd = ref_mbx(T_RMBX *pk_rmbx, ID mbxid);
```

Parameters

I/O	Parameter		Description
O	T_RMBX	*pk_rmbx;	Start address of packet used to store mailbox information
I	ID	mbxid;	Mailbox ID number

- Structure of mailbox information T_RMBX

```
typedef    struct     t_rmbx {
           VP         exinf;      /*  Extended information          */
           BOOL_ID    wtsk;       /*  Existence of waiting task     */
           T_MSG      *pk_msg;    /*  Existence of waiting message  */
           ID         keyid;      /*  Key ID number                 */
} T_RMBX;
```

Explanation

This system call stores mailbox information (extended information, existence of waiting task, etc.) for the mailbox specified by *mbxid* into the packet specified by *pk_rmbx*.
Mailbox information is described in detail below.

exinf ... Extended information

wtsk ... Existence of waiting task

- FALSE(0): No waiting task
- Value: ID number of the first task of wait queue

pk_msg ... Existence of waiting message

- NADR(–1): No waiting message
- Value: Address of the first message of wait queue

keyid ... Key ID number

- FALSE(0): No key ID number specified at creation
- Value: Key ID number

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The ref_mbx system call is not defined as CF.
E_PAR	–33	The start address of the packet used to store mailbox information is invalid (*pk_rmbx* = 0).
E_ID	–35	Invalid ID number specification (maximum number of mailboxes created < *mbxid*)
*E_NOEXS	–52	The target mailbox does not exist.
E_OACV	–66	An unauthorized ID number (*mbxid* ≤ 0) was specified.

vget_mid

Get Mailbox Identifier (–245)

Task/non-task

Overview

Acquires the mailbox ID number.

C format

```
#include      <stdrx85p.h>
ER            ercd = vget_mid(ID *p_mbxid, ID keyid);
```

Parameters

I/O	Parameter		Description
O	ID	*p_mbxid;	Address of area used to store ID number
I	ID	keyid;	Mailbox key ID number

Explanation

This system call stores the mailbox ID number specified by *keyid* in the area specified by *p_mbxid*.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The vget_mid system call is not defined as CF.
E_PAR	–33	Invalid parameter specification • Invalid key ID number specification (*keyid* = 0) • The address of the area used to store the ID number is invalid (*p_mbxid* = 0).
*E_NOEXS	–52	The target mailbox does not exist.

11.8.4 Interrupt management system calls

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

Table 11-8 lists the interrupt management system calls.

Table 11-8. Interrupt Management System Calls

System Call	Function
def_int	Registers an indirectly activated interrupt handler and cancels its registration.
ret_int	Returns from a directly activated interrupt handler.
ret_wup	Wakes up another task and returns from a directly activated interrupt handler.
ena_int	Enables the acknowledgement of maskable interrupts.
dis_int	Disables the acknowledgement of maskable interrupts.
loc_cpu	Disables the acknowledgement of maskable interrupts and dispatch processing.
unl_cpu	Enables the acknowledgement of maskable interrupts and dispatch processing.
chg_icr	Changes the interrupt control register.
ref_icr	Acquires the interrupt control register.

def_int

Define Interrupt Handler (–65)

Task/non-task

Overview

Registers an indirectly activated interrupt handler and cancels its registration.

C format

```
#include      <stdrx85p.h>
ER            ercd = def_int(UINT eintno, T_DINT *pk_dint);
```

Parameters

I/O	Parameter	Description
I	`UINT eintno;`	Interrupt request number of indirectly activated interrupt handler
I	`T_DINT *pk_dint;`	Start address of packet storing indirectly activated interrupt handler registration information

- Structure of indirectly activated interrupt handler registration information T_DINT

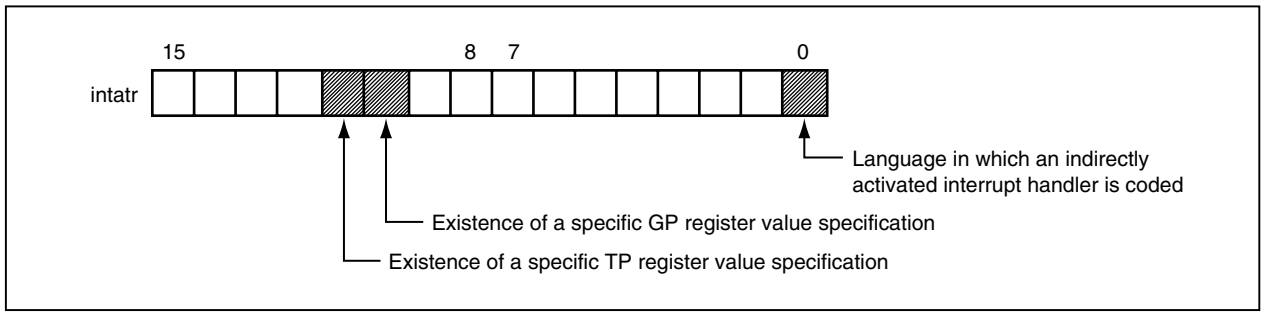
```
typedef    struct    t_dint {
           ATR       intatr;    /*  Attribute of indirectly activated interrupt handler          */
           FP        inthdr;    /*  Activation address of indirectly activated interrupt handler */
           VP        gp;        /*  Specific GP register value for indirectly activated interrupt
                                    handler                                                      */
           VP        tp;        /*  Specific TP register value for indirectly activated interrupt
                                    handler                                                      */
} T_DINT;
```

Explanation

This system call uses the information specified by *pk_dint* to register the indirectly activated interrupt handler activated upon the occurrence of the maskable interrupt with the interrupt request number specified by *eintno*. Indirectly activated interrupt handler registration information is described in detail below.

intatr ... Attribute of indirectly activated interrupt handler

- Bit 0 .. Language in which an indirectly activated interrupt handler is coded
  - TA_ASM(0): Assembly language
  - TA_HLNG(1): C language
- Bit 10 .. Existence of a specific GP register value specification
  - TA_DPID(1): Specific GP register value specified.
- Bit 11 .. Existence of a specific TP register value specification
  - TA_DPIC(1): Specific TP register value specified.



inthdr	...	Activation address of indirectly activated interrupt handler
gp	...	Specific GP register value for indirectly activated interrupt handler
tp	...	Specific TP register value for indirectly activated interrupt handler

When this system call is issued, if an indirectly activated interrupt handler corresponding to the specified interrupt request number has already been registered, this system call does not handle this as an error and newly registers the specified indirectly activated interrupt handler.

When this system call is issued, if NADR(–1) is set in the area specified by *pk_dint*, the registration of the interrupt handler specified by *eintno* is canceled.

Remarks 1. If the value of bit 10 is not 1 (TA_DPID), the contents of gp are meaningless.

2. If the value of bit 11 is not 1 (TA_DPIC), the contents of tp are meaningless.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The def_int system call is not defined as CF.
E_RSATR	–24	Invalid specification of attribute intatr
E_PAR	–33	Invalid parameter specification • Invalid interrupt request number specification (*eintno* < 0, maximum interrupt source number < *eintno*) • The start address of the packet storing indirectly activated interrupt handler registration information is invalid (*pk_dint* = 0). • Invalid specification of the activation address (inthdr = 0)

ret_int

Return from Interrupt Handler (–69)

Directly activated interrupt handler

Overview

Returns from a directly activated interrupt handler.

C format

```
#include      <stdrx85p.h>
void          ret_int();
```

Parameter

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 Pro merely queues the tasks into the wait queue and delays actual scheduling until a system call (ret_int or ret_wup) is issued to return from the directly activated interrupt handler, at which point the queued tasks are all processed in batch.

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 termination before the issuance of this system call.

2. When describing a directly activated interrupt handler in assembly language, describe the following line for return from the directly activated interrupt handler:

```
jr     _ret_int
```

However, for return using the macro RTOS_IntReturn provided in the RX850 Pro, no description is required because the ret_int system call is issued within the macro.

3. When describing an indirectly activated interrupt handler in C language, describe the following line for return from the indirectly activated interrupt handler:

```
return (TSK_NULL);
```

4. When describing an indirectly activated interrupt handler in assembly language, describe the following line for return from the indirectly activated interrupt handler:

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

Return value

None.

ret_wup

Return and Wakeup Task (–70)

Directly activated interrupt handler

Overview

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

C format

```
#include      <stdrx85p.h>
void          ret_wup(ID tskid);
```

Parameter

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

Explanation

This system call returns from a directly activated interrupt handler after the issuance of a wake-up request to the task specified by *tskid* (the wake-up request counter is incremented by 0x1).

When this system call is issued, if the target task is in the wait state (wake-up wait state), this system call changes the target task from the wake-up wait state to the ready state without issuing a wake-up request (incrementing the wake-up request counter).

If a system call (chg_pri, sig_sem, etc.) requiring task scheduling is issued from a directly activated interrupt handler, the RX850 Pro merely queues the tasks into a wait queue and delays the actual scheduling until a system call (issuing ret_wup or ret_int system call) is issued to return from the directly activated interrupt handler, at which point the queued tasks are all processed in batch.

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 issuance of this system macro.

2. In this system call, if the following types of errors occur, only processing for returning from the directly activated interrupt handler is performed.

- **Invalid ID number specification (maximum number of tasks created < *tskid*)**
- **The target task does not exist.**
- **The target task is in the dormant state.**
- **The number of wake-up requests exceeded 127.**

3. **When describing a directly activated interrupt handler in assembly language, describe the following line for return from the directly activated interrupt handler:**

```
mov    tskid, r10
jr     _ret_wup
```

However, for return using the macro RTOS_IntReturnWakeup provided in the RX850 Pro, no description is required because the ret_wup system call is issued within the macro.
If this macro is used, describe as follows.

```
RTOS_IntReturnWakeup r10 (r10 = task ID)
```

4. **When describing an indirectly activated interrupt handler in C language, describe the following line for return from the indirectly activated interrupt handler:**

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

5. **When describing an indirectly activated interrupt handler in assembly language, describe the following line for return from a wake-up and the indirectly activated interrupt handler:**

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

Return value

None.

ena_int

Enable Interrupt (–71)

Task/non-task

Overview

Enables acknowledgement of maskable interrupts.

C format

```
#include      <stdrx85p.h>
ER            ena_int();
```

Parameter

None.

Explanation

This system call allows the resumption of acknowledgement of maskable interrupts that were disabled by issuing the dis_int system call.

If a maskable interrupt occurs after the dis_int system call is issued before this ena_int system call is issued, the RX850 Pro delays switching to interrupt processing (a directly activated interrupt handler or an indirectly activated interrupt handler) until the ena_int system call is issued.

Caution This system call does not queue resume requests. Therefore, if this system call has been issued already and acknowledgement of maskable interrupts has been enabled, no processing is executed and it is not treated as an error.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The ena_int system call is not defined as CF.

dis_int

Disable Interrupt (–72)

Task/non-task

Overview

Disables acknowledgement of maskable interrupts.

C format

```
#include      <stdrx85p.h>
ER            dis_int();
```

Parameter

None.

Explanation

This system call disables the acknowledgement of maskable interrupts.

This disables the acknowledgement of maskable interrupts before the ena_int system call is issued.

If a maskable interrupt occurs after the dis_int system call is issued before this ena_int system call is issued, the RX850 Pro delays switching to interrupt processing (a directly activated interrupt handler or an indirectly activated interrupt handler) until the ena_int system call is issued.

Caution This system call does not queue disable requests. Therefore, if this system call has been issued already and acknowledgement of maskable interrupts has been disabled, no processing is executed and it is not treated as an error.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The dis_int system call is not defined as CF.

loc_cpu

Lock CPU (–8)

Task

Overview

Disables the acknowledgement of maskable interrupts and dispatch processing.

C format

```
#include      <stdrx85p.h>
ER            ercd = loc_cpu();
```

Parameter

None.

Explanation

This system call disables the acknowledgement of maskable interrupts and dispatch processing (task scheduling). Therefore, for the period of time from the issuance of this system call to the issuance of the unl_cpu, there is no transfer of control to another handler or task.

If a maskable interrupt occurs after this system call is issued but before the unl_cpu system call is issued, the RX850 Pro delays processing for the interrupt (interrupt handler) until the unl_cpu system call is issued. If a system call (chg_pri, sig_sem, etc.) requiring task scheduling is issued, the RX850 Pro merely queues the tasks into a wait queue and delays the actual scheduling until the unl_cpu system call is issued, at which point all the tasks are processed in batch.

Caution This system call does not queue disable requests. Therefore, if this system call has been issued already and acknowledgement of maskable interrupts and dispatch processing has been disabled, no processing is executed and it is not treated as an error.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The loc_cpu system call is not defined as CF.
E_CTX	–69	The loc_cpu system call was issued from a non-task.

unl_cpu

Unlock CPU (–7)

Task

Overview

Enables the acknowledgement of maskable interrupts and dispatch processing.

C format

```
#include      <stdrx85p.h>
ER            ercd = unl_cpu();
```

Parameter

None.

Explanation

This system call allows the resumption of acknowledgement of maskable interrupts and dispatch processing (task scheduling) disabled by issuing 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 Pro delays processing for the interrupt (interrupt handler) until this system call is issued. If a system call (chg_pri, sig_sem, etc.) requiring task scheduling is issued, the RX850 Pro merely queues the tasks into a wait queue and delays actual scheduling until the unl_cpu system call is issued, at which point all the tasks are processed in batch.

Remark Dispatch processing that was disabled by the issuance of the dis_dsp system call is reenabled by this system call.

Caution This system call does not queue resume requests. Therefore, if this system call has been issued already and acknowledgement of maskable interrupts and dispatch processing has resumed, no processing is executed and it is not treated as an error.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The unl_cpu system call is not defined as CF.
E_CTX	–69	The unl_cpu system call was issued from a non-task.

chg_icr

Change Interrupt Control Register (–67)

Task/non-task

Overview

Changes the contents of the 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 interrupt request flag ICR_CLRINT (0x20): No interrupt request Specification of interrupt mask flag ICR_CLRMSK (0x10): Enables interrupt processing ICR_SETMSK (0x40): Disables interrupt processing Specification of changing interrupt priority order ICR_CHGLVL (0x08): Changes interrupt priority order Specification of interrupt priority order Value (0 to 7): Interrupt priority order

Explanation

This system call changes the contents of the interrupt control register specified by *eintno* to the value specified by *icrcmd*. The *icrcmd* specification formats are as follows:

- *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 order of the interrupt control register to the value specified by “Value”. The value “0” corresponds to level 0 and value “7” to level 7.

Remark Specify the value calculated by [(the exceptional code of the specified interrupt request number – 0x80) / 0x10] for the interrupt request number *eintno*.

Caution When the RX850 Pro is operated on the V850E core, even if the system call chg_icr is issued, the desired interrupt control register may not operate. In the RX850 Pro, the interrupt control register address is calculated from the interrupt source number. However, in the V850E core, the correct register address cannot be obtained since the alignment of the interrupt source numbers and interrupt control registers differs from other V850 Series products. Therefore, use of the system call chg_icr is restricted. For manipulating the interrupt control register via an application, directly manipulate the register without using this system call.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The chg_icr system call is not defined as CF.
E_PAR	–33	Invalid parameter specification • Invalid specification of interrupt request number (*eintno* < 0, maximum interrupt source number < *eintno*) • Invalid specification of interrupt request flag (*eintno* < 0, maximum interrupt source number < *eintno*) • (ICR_CLRMSK | ICR_SETMSK) is specified as an interrupt request flag.

ref_icr

Refer Interrupt Control Register Status (–68)

Task/non-task

Overview

Acquires the contents of the interrupt control register.

C format

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

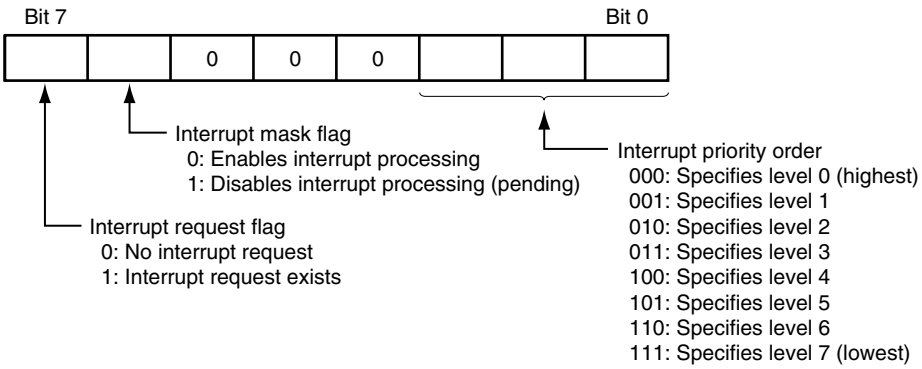
Parameters

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

Explanation

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

The following figure shows the contents of the acquired interrupt control register:



Remark Specify the value calculated by [(the exceptional code of the specified interrupt request number − 0x80) / 0x10] for the interrupt request number *eintno*.

Caution When the RX850 Pro is operated on the V850E core, even if the system call ref_icr is issued, the desired interrupt control register may not operate. In the RX850 Pro, the interrupt control register address is calculated from the interrupt source number. However, in the V850E core, the correct register address cannot be obtained since the alignment of the interrupt source numbers and interrupt control registers differs from other V850 Series products. Therefore, use of the system call ref_icr is restricted. For manipulating the interrupt control register via an application, directly manipulate the register without using this system call.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The ref_icr system call is not defined as CF.
E_PAR	–33	Invalid parameter specification • The address of the area used to store the contents of interrupt control register (*p_regptn*) is 0. • Invalid specification of interrupt request number (*eintno* < 0, maximum interrupt source number < *eintno*)

11.8.5 Memory pool management system calls

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

Table 11-9 lists the memory pool management system calls.

Table 11-9. Memory Pool Management System Calls

System Call	Function
cre_mpl	Creates a memory pool.
del_mpl	Deletes a memory pool.
get_blk	Acquires a memory block.
pget_blk	Acquires a memory block (polling).
tget_blk	Acquires a memory block (with timeout).
rel_blk	Returns a memory block.
ref_mpl	Acquires memory pool information.
vget_pid	Acquires memory pool ID number.

cre_mpl

Create Variable-size Memory Pool (–137)

Task

Overview

Creates a memory pool.

C format

- When an ID number is specified

```
#include        <stdrx85p.h>
ER              ercd = cre_mpl(ID mplid, T_CMPL *pk_cmpl);
```

- When an ID number is not specified

```
#include        <stdrx85p.h>
ER              ercd = cre_mpl(ID_AUTO, T_CMPL *pk_cmpl, ID *p_mplid);
```

Parameters

I/O	Parameter		Description
I	ID	*mplid*;	Memory pool ID number
I	T_CMPL	**pk_cmpl*;	Start address of packet containing memory pool creation information
O	ID	**p_mplid*;	Address of area used to store ID number

- Structure of memory pool creation information T_CMPL

```
typedef    struct    t_cmpl {
           VP        exinf;     /*  Extended information          */
           ATR       mplatr;    /*  Memory pool attribute         */
           INT       mplsz;     /*  Memory pool size              */
           ID        keyid;     /*  Memory pool key ID number     */
} T_CMPL;
```

Explanation

The RX850 Pro provides two types of interfaces for memory pool creation: one in which an ID number must be specified for memory pool creation, and one in which an ID number is not specified.

- When an ID number is specified
  A memory pool having the ID number specified by *mplid* is created based on the information specified by *pk_cmpl*.

- When an ID number is not specified
  A memory pool is created based on the information specified by *pk_cmpl*.
  An ID number is allocated by the RX850 Pro and the allocated ID number is stored in the area specified by *p_mplid*.

Memory pool creation information is described in detail below.

exinf ... Extended information

exinf is an area used for storing user-specific information for the specified memory pool. The user can use this area as required.

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

mplatr ... Memory pool attribute

Bit 0 .. Method of queuing to a wait queue

TA_TPRI(0): Priority order

TA_TFIFO(1): FIFO order

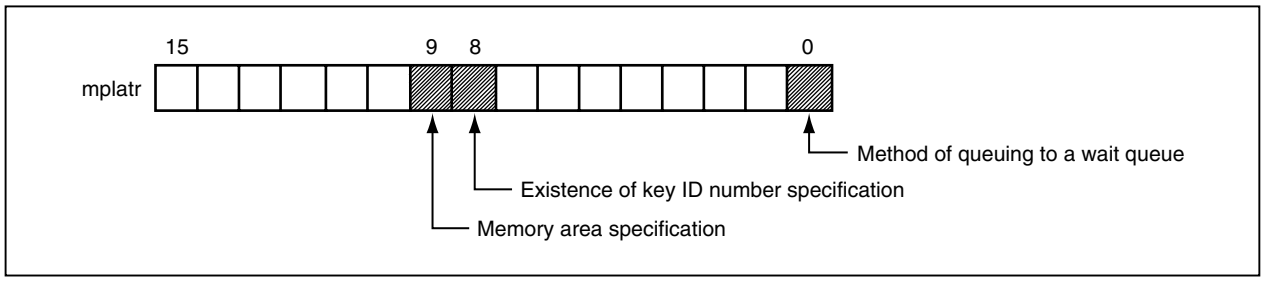
Bit 8 .. Existence of key ID number specification

TA_KEYID(1): Key ID number specified

Bit 9 .. Memory area specification

TA_UPOL0(0): Secures the memory pool area from user memory area 0.

TA_UPOL1(1): Secures the memory pool area from user memory area 1.



mplsz ... Memory pool size (unit: byte)

keyid ... Memory pool key ID number

Remark If the value of bit 8 is not 1 (TA_KEYID) in mplatr, the contents of keyid are meaningless.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The cre_mpl system call is not defined as CF.
*E_NOMEM	–10	A memory pool management block or memory pool area cannot be allocated.
E_RSATR	–24	Invalid specification of attribute mplatr
E_PAR	–33	Invalid parameter specification • The start address of the packet storing memory pool creation information is invalid (*p_mplid* = 0). • Invalid size specification (mplsz ≤ 0) • Invalid key ID number specification (keyid = 0) (at TA_KEYID attribute specified) • The address of the area used to store the ID number is invalid (*p_mplid* = 0) (for creation without specifying the ID number)
E_ID	–35	Invalid ID number specification (maximum number of memory pools created < *mplid*)
*E_OBJ	–63	A memory pool having the specified ID number has already been created.
E_OACV	–66	An unauthorized ID number (*mplid* ≤ 0) was specified.
E_CTX	–69	The cre_mpl system call was issued from a non-task.

del_mpl

Delete Variable-size Memory Pool (–138)

Task

Overview

Deletes a memory pool.

C format

```
#include      <stdrx85p.h>
ER            ercd = del_mpl(ID mplid);
```

Parameter

I/O	Parameter	Description
I	ID mplid;	Memory pool ID number

Explanation

This system call deletes the memory pool specified by *mplid*.

The target memory pool is released from the control of the RX850 Pro.

The task released from the wait state (memory block wait state) by this system call has E_DLT returned as the return value of the system call (get_blk or tget_blk) that instigated the transition to the wait state.

If this system call is issued when the task acquires a memory block that the target memory pool manages, the memory block is also released from the control of the RX850 Pro. Accordingly, the contents of the acquired memory block are undefined.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The del_mpl system call is not defined as CF.
E_ID	–35	Invalid ID number specification (maximum number of memory pools created < *mplid*)
*E_NOEXS	–52	The target memory pool does not exist.
E_OACV	–66	An unauthorized ID number (*mplid* ≤ 0) was specified.
E_CTX	–69	The del_mpl system call was issued from a non-task.

get_blk

Get Variable-size Memory Block (–141)

Task

Overview

Acquires a memory block.

C format

```
#include      <stdrx85p.h>
ER            ercd = get_blk(VP *p_blk, ID mplid, INT blksz);
```

Parameters

I/O	Parameter		Description
O	VP	*p_blk;	Address of area used to store start address of memory block
I	ID	mplid;	Memory pool ID number
I	INT	blksz;	Memory block size (unit: bytes)

Explanation

This system call acquires a memory block of the size specified by *blksz* from the memory pool specified by *mplid* and stores its start address in the area specified by *p_blk*.

If no memory block can be acquired from the target memory pool (when there is no free area of the requested size) upon the issuance of this system call, this system call places the task in the wait queue of the target memory pool before changing its state from the run state to the wait state (memory block wait state).

The memory block wait state is released when a memory block that satisfies the requested size is released by the rel_blk system call or upon the issuance of the del_mpl or rel_wai system call, and the task returns to the ready state.

Caution The RX850 Pro does not clear the memory upon acquiring a memory block. Accordingly, the contents of the acquired memory block are undefined.

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

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The get_blk system call is not defined as CF.
E_PAR	–33	Invalid parameter specification • The address of the area used to store the start address of the memory block is invalid (*p_blk* = 0). • Invalid specification of memory block size (*blksz* ≤ 0)
E_ID	–35	Invalid ID number specification (maximum number of memory pools created < *mplid*)
*E_NOEXS	–52	The target memory pool does not exist.
E_OACV	–66	An unauthorized ID number (*mplid* ≤ 0) was specified.
E_CTX	–69	Context error • The get_blk system call was issued from a non-task. • The get_blk system call was issued in the dispatch disabled state.
*E_DLT	–81	The target memory pool was deleted by the del_mpl system call.
*E_RLWAI	–86	The memory block wait state was forcibly released by the rel_wai system call.

pget_blk

Poll and Get Variable-size Memory Block (–104)

Task/non-task

Overview

Acquires a memory block (polling).

C format

```
#include      <stdrx85p.h>
ER            ercd = pget_blk(VP *p_blk, ID mplid, INT blksz);
```

Parameters

I/O	Parameter		Description
O	VP	*p_blk;	Address of area used to store start address of memory block
I	ID	mplid;	Memory pool ID number
I	INT	blksz;	Memory block size (unit: bytes)

Explanation

This system call acquires a memory block of the size specified by *blksz* from the memory pool specified by *mplid* and stores its start address in the area specified by *p_blk*.

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

Caution The RX850 Pro does not clear the memory upon acquiring a memory block. Accordingly, the contents of the acquired memory block are undefined.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The pget_blk system call is not defined as CF.
E_PAR	–33	Invalid parameter specification • The address of the area used to store the start address of a memory block is invalid (*p_blk* = 0). • Invalid specification of memory block size (*blksz* ≤ 0)
E_ID	–35	Invalid ID number specification (maximum number of memory pools created < *mplid*)
*E_NOEXS	–52	The target memory pool does not exist.
E_OACV	–66	An unauthorized ID number (*mplid* ≤ 0) was specified.
*E_TMOUT	–85	There is no free space in the target memory pool.

tget_blk

Get Variable-size Memory Block with Timeout (–168)

Task

Overview

Acquires a memory block (with timeout).

C format

```
#include     <stdrx85p.h>
ER           ercd =  tget_blk(VP *p_blk, ID mplid, INT blksz, TMO tmout);
```

Parameters

I/O	Parameter		Description
O	VP	*p_blk;	Address of area used to store start address of memory block
I	ID	mplid;	Memory pool ID number
I	INT	blksz;	Memory block size (unit: bytes)
I	TMO	tmout;	Wait time (unit: ms) TMO_POL(0): Quick return TMO_FEVR(–1): Permanent wait Value: Wait time

Explanation

This system call acquires a memory block of the size specified by *blksz* from the memory pool specified by *mplid* and stores its start address in the area specified by *p_blk*.

If a memory block cannot be acquired from the target memory pool (when there is no free area of the requested size) when this system call is issued, this system call places the task in the wait queue of the target memory pool before changing it from the run state to the wait state (memory block wait state).

The memory block wait state is released when the wait time specified by *tmout* elapses, when a memory block that satisfies the requested size is released by the rel_blk system call, or when the del_mpl or rel_wai system call is issued. The task then returns to the ready state.

Caution The RX850 Pro does not clear the memory upon acquiring a memory block. Accordingly, the contents of the acquired memory block are undefined.

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

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The tget_blk system call is not defined as CF.
E_PAR	–33	Invalid parameter specification • The address of the area used to store the start address of the memory block is invalid (*p_blk* = 0). • Invalid specification of memory block size (*blksz* ≤ 0)
E_ID	–35	Invalid ID number specification (maximum number of memory pools created < *mplid*)
*E_NOEXS	–52	The target memory pool does not exist.
E_OACV	–66	An unauthorized ID number (*mplid* ≤ 0) was specified.
E_CTX	–69	Context error • The tget_blk system call was issued from a non-task. • The tget_blk system call was issued in the dispatch disabled state.
*E_DLT	–81	The target memory pool was deleted by the del_mpl system call.
*E_TMOUT	–85	Timeout elapsed.
*E_RLWAI	–86	The memory block wait state was forcibly released by the rel_wai system call.

rel_blk

Release Variable-size Memory Block (–143)

Task/non-task

Overview

Returns a memory block.

C format

```
#include      <stdrx85p.h>
ER            ercd = rel_blk(ID mplid, VP blk);
```

Parameters

I/O	Parameter		Description
I	ID	*mplid;*	Memory pool ID number
I	VP	*blk;*	Start address of memory block

Explanation

This system call returns the memory block specified by *blk* to the memory pool specified by *mplid*.
If the size of the returned memory block satisfies the size requested by the (first) task queuing in the target memory pool's wait queue when this system call is issued, the memory block is transferred to that task.
The relevant task is consequently removed from the wait queue, and changes from the wait state (memory block wait state) to the ready state, or from the wait-suspend state to the suspend state.

Cautions 1. The contents of a returned memory block are not cleared by the RX850 Pro. Thus, the contents of a memory block may be undefined when that memory block is returned.

★ **2. The RX850 Pro includes two different specifications for the rel_blk system call.**

(1) When a memory block is returned by a rel_blk system call, if the first four bytes of the memory block are not filled with zeros, the return value E_OBJ is used for termination instead of returning the memory block.

(2) When the rel_blk system call is issued, the memory block is returned even if the first four bytes of the memory block are not filled with zeros (return value = E_OK).

The first specification applies when the memory block is used as a mailbox's message area, and this is the specification that has been used for the rel_blk system call as it has been implemented thus far in the RX850 Pro.

When the memory block is used as a mailbox's message area, the first four bytes serve as the link area for the message's wait queue. In other words, if messages are queued in the mailbox, when the rel_blk system call is issued and the memory block must be returned, in which case it is the message area linked to the queue that is returned. To prevent this, the specification requires the first four bytes that comprise the link area to be filled with zeros, otherwise it will be recognized as the memory block used as the message area and the return value E_OBJ will be used for termination instead of returning the memory block. Under this specification, the first four bytes must be cleared to zero in order to use rel_blk to return the memory block.
These specifications of rel_blk are stored in separate libraries so that one or the other rel_blk specification can be used. Link to the library of the rel_blk specification to be used.

(1) Library containing rel_blk that requires zero-clearing of first four bytes of memory block → librxp.a

(2) Library containing rel_blk that does not require zero-clearing of first four bytes of memory block → librxpm.a

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

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The rel_blk system call is not defined as CF.
E_PAR	–33	Invalid parameter specification • Invalid specification of the start address of a memory block (*blk* = 0) • The memory pool specified when acquired differs from that specified upon the issuance of the rel_blk system call.
E_ID	–35	Invalid ID number specification (maximum number of memory pools created < *mplid*)
*E_NOEXS	–52	The target memory pool does not exist.
E_OBJ	–63	A value other than 0x0 is placed in the first four bytes of the memory block to be returned. • This return value is returned when returning the memory block used as a message area.
E_OACV	–66	An unauthorized ID number (*mplid* ≤ 0) was specified.

ref_mpl

Refer Variable-size Memory Pool Status (–140)

Task/non-task

Overview

Acquires memory pool information.

C format

```
#include      <stdrx85p.h>
ER            ercd = ref_mpl(T_RMPL *pk_rmpl, ID mplid);
```

Parameters

I/O	Parameter		Description
O	T_RMPL	*pk_rmpl;	Start address of packet used to store memory pool information
I	ID	mplid;	Memory pool ID number

- Structure of memory pool information T_RMPL

```
typedef    struct     t_rmpl {
           VP         exinf;     /*  Extended information                          */
           BOOL_ID    wtsk;      /*  Existence of waiting task                     */
           INT        frsz;      /*  Total size of free area                       */
           INT        maxsz;     /*  Maximum memory block size that can be acquired */
           ID         keyid;     /*  Key ID number                                 */
} T_RMPL;
```

Explanation

This system call stores the memory pool information (extended information, existence of waiting tasks, etc.) for the memory pool specified by *mplid* in the packet specified by *pk_rmpl*.
Memory pool information is described in detail below.

exinf ... Extended information
wtsk ... Existence of waiting task
  FALSE(0): No waiting task
  Value: ID number of first task in the wait queue
frsz ... Total size of free area (unit: bytes)
maxsz ... Maximum memory block size that can be acquired (unit: bytes)
keyid ... Key ID number
  FALSE(0): No specification for key ID number at generation
  Value: Key ID number

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The ref_mpl system call is not defined as CF.
E_PAR	–33	The start address of the packet used to store memory pool information is invalid (*pk_rmpl* = 0).
E_ID	–35	Invalid ID number specification (maximum number of memory pools created < *mplid*)
*E_NOEXS	–52	The target memory pool does not exist.
E_OACV	–66	An unauthorized ID number (*mplid* ≤ 0) was specified.

vget_pid

Get Variable-size Memory Pool Identifier (–242)

Task/non-task

Overview

Acquires the memory pool ID number.

C format

```
#include      <stdrx85p.h>
ER            ercd = vget_pid(ID *p_mplid, ID keyid);
```

Parameters

I/O	Parameter		Description
O	ID	*p_mplid;	Address of area used to store ID number
I	ID	keyid;	Memory pool key ID number

Explanation

This system call stores the memory pool ID number specified by *keyid* in the area specified by *p_mplid*.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The vget_pid system call is not defined as CF.
E_PAR	–33	Invalid parameter specification • Invalid key ID number specification (*keyid* = 0) • The address of the area used to store the ID number is invalid (*p_mplid* = 0).
*E_NOEXS	–52	The target memory pool does not exist.

11.8.6 Time management system calls

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

Table 11-10 lists the time management system calls.

Table 11-10. Time Management System Calls

System Call	Function
set_tim	Sets the system clock.
get_tim	Acquires the time from the system clock.
dly_tsk	Changes the task to the timeout wait state.
def_cyc	Registers a cyclically activated handler or cancels its registration.
act_cyc	Controls the activity state of a cyclically activated handler.
ref_cyc	Acquires cyclically activated handler information.

set_tim

Set Time (–83)

Task/non-task

Overview

Sets the system clock.

C format

```
#include      <stdrx85p.h>
ER            ercd = set_tim(SYSTIME *pk_tim);
```

Parameter

I/O	Parameter	Description
I	SYSTIME *pk_tim;	Start address of packet storing time

- Structure of system clock SYSTIME

★

```
typedef    struct     t_systime {
           UW         ltime;     /*  Time (lower 32 bits)      */
           H          utime;     /*  Time (higher 16 bits)     */
} SYSTIME;
```

Explanation

This system call sets the system clock to the time specified by *pk_tim*.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The set_tim system call is not defined as CF.
E_PAR	–33	The start address of the packet storing time is invalid (*pk_tim* = 0).

get_tim

Get Time (–84)

Task/non-task

Overview

Acquires the time from the system clock.

C format

```
#include      <stdrx85p.h>
ER            ercd = get_tim(SYSTIME *pk_tim);
```

Parameter

I/O	Parameter	Description
O	SYSTIME *pk_tim;	Start address of packet storing time

- Structure of system clock SYSTIME

★
```
typedef    struct     t_systime {
           UW         ltime;     /*  Time (lower 32 bits)      */
           H          utime;     /*  Time (higher 16 bits)     */
} SYSTIME;
```

Explanation

This system call sets the current system clock time in the packet specified by *pk_tim*.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The get_tim system call is not defined as CF.
E_PAR	–33	The start address of the packet storing time is invalid (*pk_tim* = 0).

dly_tsk

Delay Task (–85)

Task

Overview

Changes the task to the timeout wait state.

C format

```
#include      <stdrx85p.h>
ER            ercd = dly_tsk(DLYTIME dlytim);
```

Parameter

I/O	Parameter	Description
I	`DLYTIME dlytim;`	Delay time (unit: ms)

Explanation

This system call changes the state of the task from the run state to the wait state (timeout wait state) for the delay time specified by *dlytim*.

The timeout wait state is released upon the elapse of the delay specified by *dlytim* or when the rel_wai system call is issued. The task then returns to the ready state.

Caution The timeout wait state is released by neither the wup_tsk or ret_wup system call.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The dly_tsk system call is not defined as CF.
E_PAR	–33	Invalid specification of delay time (*dlytim* < 0)
E_CTX	–69	Context error • The dly_tsk system call was issued from a non-task. • The dly_tsk system call was issued in the dispatch disabled state.
*E_RLWAI	–86	The timeout wait state was forcibly released by the rel_wai system call.

def_cyc

Define Cyclic Handler (–90)

Task/non-task

Overview

Registers a cyclically activated handler or cancels its registration.

C format

```
#include      <stdrx85p.h>
ER            ercd = def_cyc(HNO cycno, T_DCYC *pk_dcyc);
```

Parameters

I/O	Parameter		Description
I	HNO	*cycno;*	Specification number of cyclically activated handler
I	T_DCYC	**pk_dcyc;*	Start address of packet storing cyclically activated handler registration information

- Structure of cyclically activated handler registration information T_DCYC

```
typedef   struct    t_dcyc {
          VP        exinf;    /*  Extended information                                  */
          ATR       cycatr;   /*  Attribute of cyclically activated handler             */
          FP        cychdr;   /*  Activation address of cyclically activated handler    */
          UINT      cycact;   /*  Initial activity state of cyclically activated handler*/
          CYCTIME   cyctim;   /*  Activation time interval of cyclically activated handler*/
          VP        gp;       /*  Specific GP register value of cyclically activated handler*/
          VP        tp;       /*  Specific TP register value of cyclically activated handler*/
} T_DCYC;
```

Explanation

This system call uses the information specified by *pk_dcyc* to register the cyclically activated handler having the specification number specified by *cycno*.

Cyclically activated handler registration information is described in detail below.

exinf ... Extended information

exinf is an area used for storing user-specific information on the specified cyclically activated handler. The user can use this area as required.

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

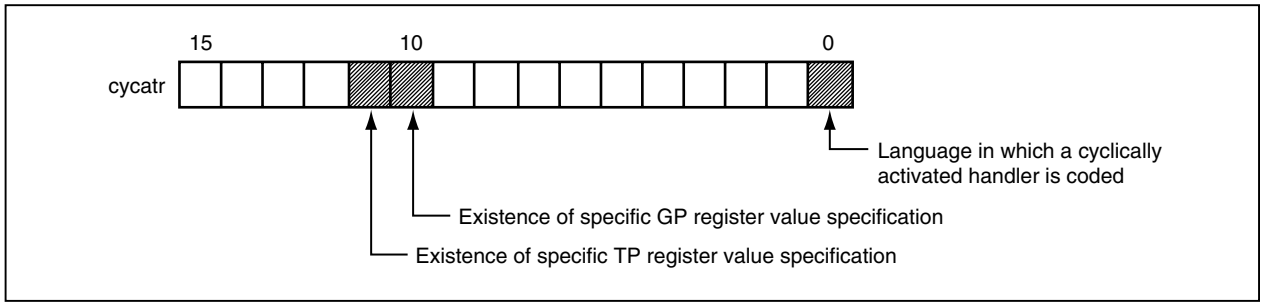
cycatr ... Attribute of cyclically activated handler

Bit 0 .. Language in which the cyclically activated handler is coded

TA_ASM(0): Assembly language

TA_HLNG(1): C language

Bit 10 .. Existence of specific GP register value specification
TA_DPID(1): Specific GP register value specified.
Bit 11 .. Existence of specific TP register value specification
TA_DPIC(1): Specific TP register value specified.



cychdr	...	Activation address of cyclically activated handler
cycact	...	Initial activity state of cyclically activated handler
		TCY_OFF(0): The initial activity state is OFF
		TCY_ON(1): The initial activity state is ON
cyctim	...	Activation time interval of cyclically activated handler (unit: basic clock cycles)
gp	...	Specific GP register value for cyclically activated handler
tp	...	Specific TP register value for cyclically activated handler

When this system call is issued, if a cyclically activated handler corresponding to the target specification number is already registered, this system call does not handle this as an error and newly registers the specified cyclically activated handler.

If this system call is issued with NADR(–1) set in the area specified by *pk_dcyc*, the registration of the cyclically activated handler specified by *cycno* is canceled.

Remarks 1. If the value of bit 10 is not 1 (TA_DPID) in cycatr, the contents of gp are meaningless.
2. If the value of bit 11 is not 1 (TA_DPIC) in cycatr, the contents of tp are meaningless.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The def_cyc system call is not defined as CF.
E_RSATR	–24	Invalid specification of attribute cycatr
E_PAR	–33	Invalid parameter specification • Invalid specification of specification number (*cycno* ≤ 0, maximum number of cyclically activated handlers that can be registered < *cycno*) • The start address of the packet storing cyclically activated handler registration information is invalid (*pk_dcyc* = 0). • Invalid specification of activation address (cychdr = 0) • Invalid specification of initial activity state cycact • Invalid specification of activation time interval (cyctim ≤ 0)

act_cyc

Activate Cyclic Handler (–94)

Task/non-task

Overview

Controls the activity state of a cyclically activated handler.

C format

```
#include      <stdrx85p.h>
ER            ercd = act_cyc(HNO cycno, UINT cycact);
```

Parameters

I/O	Parameter		Description
I	HNO	*cycno;*	Specification number of cyclically activated handler
I	UINT	*cycact;*	Specification of activity state and cycle counter TCY_OFF(0): Changes the activity state to the OFF state. TCY_ON(1): Changes the activity state to the ON state. TCY_INI(2): Initializes the cycle counter.

Explanation

This system call changes the activity state of the cyclically activated handler specified by *cycno* to the state specified by *cycact*. The specification format of *cycact* is described below.

- *cycact* = TCY_OFF
  Changes the activity state of the target cyclically activated handler to the OFF state.
  Even when the activation time is reached, the target cyclically activated handler is not activated.

  Caution Even when the activity state of the cyclically activated handler is OFF, the RX850 Pro increments the cycle counter.

- *cycact* = TCY_ON
  Changes the activity state of the target cyclically activated handler to the ON state.
  When the activation time is reached, the target cyclically activated handler is activated.

- *cycact* = TCY_INI
  Initializes the cycle counter of the target cyclically activated handler.

- *cycact* = (TCY_ON|TCY_INI)
  Changes the activity state of the target cyclically activated handler to the ON state before initializing the cycle counter.
  When the activation time is reached, the target cyclically activated handler is activated.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The act_cyc system call is not defined as CF.
E_PAR	–33	Invalid parameter specification • The specification number of the cyclically activated handler is invalid (cycno ≤ 0, maximum number of cyclically activated handlers that can be registered < cycno). • Invalid specification of activity state or cycle counter cycact
*E_NOEXS	–52	The target cyclically activated handler is not registered.

ref_cyc

Refer Cyclic Handler Status (–92)

Task/non-task

Overview

Acquires cyclically activated handler information.

C format

```
#include      <stdrx85p.h>
ER            ercd = ref_cyc(T_RCYC *pk_rcyc, HNO cycno);
```

Parameters

I/O	Parameter		Description
O	T_RCYC	*pk_rcyc;	Start address of packet used to store cyclically activated handler information
I	HNO	cycno;	Specification number of cyclically activated handler

- Structure of cyclically activated handler information T_RCYC

```
typedef    struct     t_rcyc {
           VP         exinf;     /*  Extended information        */
           CYCTIME    lfttim;    /*  Remaining time              */
           UINT       cycact;    /*  Current activity state      */
} T_RCYC;
```

Explanation

This system call stores the cyclically activated handler information (extended information, remaining time, etc.) of the cyclically activated handler specified by *cycno* in the packet specified by *pk_rcyc*.
Cyclically activated handler information is described in detail below.

exinf … Extended information

lfttim … Time remaining until the cyclically activated handler is next activated (unit: basic clock cycles)

cycact … Current activity state

TCY_OFF(0): Activity state is OFF.

TCY_ON(1): Activity state is ON.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The ref_cyc system call is not defined as CF.
E_PAR	–33	Invalid parameter specification • The start address of the packet used to store cyclically activated handler information is invalid (*pk_rcyc* = 0). • The specification number of the cyclically activated handler is invalid (*cycno* ≤ 0, maximum number of cyclically activated handlers that can be registered < *cycno*).
*E_NOEXS	–52	The target cyclically activated handler is not registered.

11.8.7 System management system calls

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

Table 11-11 lists the system management system calls.

Table 11-11. System Management System Calls

System Call	Function
get_ver	Acquires RX850 Pro version information.
ref_sys	Acquires system information.
def_svc	Registers an extended SVC handler or cancels its registration.
viss_svc	Calls an extended SVC handler.

get_ver

Get Version Information (–16)

Task/non-task

Overview

Acquires RX850 Pro version information.

C format

```
#include      <stdrx85p.h>
ER            ercd = get_ver(T_VER *pk_ver);
```

Parameter

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 Pro version information (OS maker, OS format, etc.) in the packet specified by *pk_ver*.

Version information is described in detail below.

maker ... OS maker

H'000d: NEC

id ... OS format

H'0000: Not used

spver ... Specification version

H'5302: μITRON3.0 Ver. 3.02

prver ... OS product version

H'0300: RX850 Pro Ver. 3.00

prno[4]	...	Product number/product management information
		Undefined: Serial number of delivery product (each unit has a unique number)
cpu	...	CPU information
		H'0d37: μPD703100
var	...	Variation descriptor
		H'c000: μITRON level E, file not supported

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The get_ver system call is not defined as CF.
E_PAR	–33	The start address of the packet used to store version information is invalid (*pk_ver* = 0).

ref_sys

Refer System Status (–12)

Task/non-task

Overview

Acquires system information.

C format

```
#include      <stdrx85p.h>
ER            ercd = ref_sys(T_RSYS *pk_rsys);
```

Parameter

I/O	Parameter	Description
O	`T_RSYS *pk_rsys;`	Start address of packet used to store system information

- Structure of system information T_RSYS

```
typedef    struct    t_rsys {
           INT       sysstat;     /*  System state     */
} T_RSYS;
```

Explanation

This system call stores the current value of dynamically-changing system information (system state) in the packet specified by *pk_rsys*.
System information is described in detail below.

sysstat … System state

TSS_TSK(0):	Task processing is being performed. Dispatch processing is enabled.
TSS_DDSP(1):	Task processing is being performed. Dispatch processing is disabled.
TSS_LOC(3):	Task processing is being performed. The acknowledgement of maskable interrupts and dispatch processing is disabled.
TSS_INDP(4):	Processing of a non-task (interrupt handler, cyclically activated handler, etc.) is being performed.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The ref_sys system call is not defined as CF.
E_PAR	–33	The start address of the packet used to store system information is invalid (*pk_rsys* = 0).

def_svc

Define Supervisor Call Handler (–9)

Task/non-task

Overview

Registers an extended SVC handler or cancels its registration.

C format

```
#include      <stdrx85p.h>
ER            ercd = def_svc(FN s_fncd, T_DSVC *pk_dsvc);
```

Parameters

I/O	Parameter		Description
I	FN	*s_fncd;*	Extended function code of extended SVC handler
I	T_DSVC	**pk_dsvc;*	Start address of packet storing the extended SVC handler registration information

- Structure of extended SVC handler registration information T_DSVC

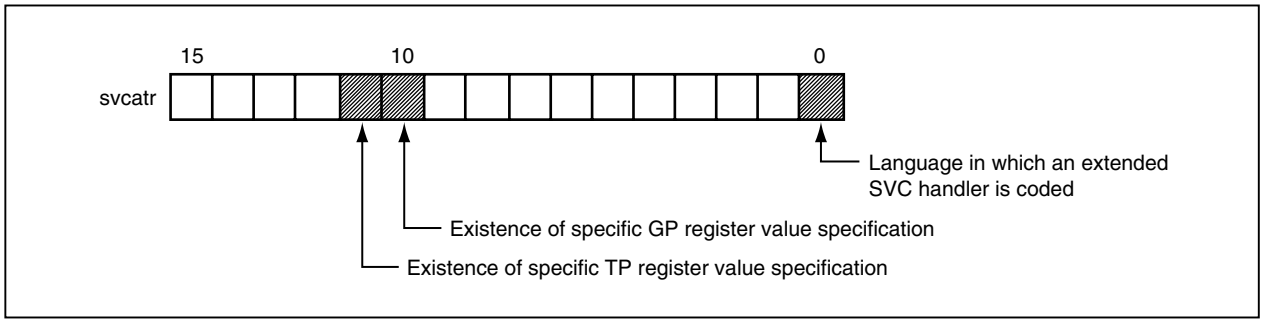
```
typedef    struct    t_dsvc {
           ATR       svcatr;    /*  Attribute of extended SVC handler                  */
           FP        svchdr;    /*  Activation address of extended SVC handler         */
           VP        gp;        /*  Specific GP register value for extended SVC handler */
           VP        tp;        /*  Specific TP register value for extended SVC handler */
} T_DSVC;
```

Explanation

This system call uses information specified by *pk_dsvc* to register the extended SVC handler having the extended function code specified by *s_fncd*.
Extended SVC handler registration information is described in detail below.

svcatr ... Attribute of extended SVC handler

- Bit 0 ... Language in which the extended SVC handler is coded
  - TA_ASM(0): Assembly language
  - TA_HLNG(1): C language
- Bit 10 ... Existence of specific GP register value specification
  - TA_DPID(1): Specific GP register value specified.
- Bit 11 ... Existence of specific TP register value specification
  - TA_DPIC(1): Specific TP register value specified.



svchdr	...	Activation address of extended SVC handler
gp	...	Specific GP register value of extended SVC handler
tp	...	Specific TP register value of extended SVC handler

When this system call is issued, if an extended SVC handler corresponding to the target extended function code has already been registered, this system call does not handle this as an error and newly registers the specified extended SVC handler.

When this system call is issued, if NADR(–1) is set in the area specified by *pk_dsvc*, the registration of the extended SVC handler specified by *s_fncd* is canceled.

Remarks 1. If the value of bit 10 is not 1 (TA_DPID) in svcatr, the contents of gp are meaningless.

2. If the value of bit 11 is not 1 (TA_DPIC) in svcatr, the contents of tp are meaningless.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The def_svc system call is not defined as CF.
E_RSATR	–24	Invalid specification of attribute svcatr
E_PAR	–33	Invalid parameter specification • Invalid specification of extended function code (*s_fncd* ≤ 0, maximum number of extended SVC handlers that can be registered < *s_fncd*) • The start address of the packet storing the extended SVC handler registration information is invalid (*pk_dsvc* = 0). • Invalid specification of activation address (svchdr = 0)

viss_svc

Issued Supervisor Call Handler (–250)

Task/non-task

Overview

Calls an extended SVC handler.

C format

```
#include      <stdrx85p.h>
ER            ercd = viss_svc(FN s_fncd, VW prm1, VW prm2, VW prm3);
```

Parameters

I/O	Parameter		Description
I	FN	*s_fncd;*	Extended function code of extended SVC handler
I	VW	*prm1;*	Parameter 1 passed to extended SVC handler
I	VW	*prm2;*	Parameter 2 passed to extended SVC handler
I	VW	*prm3;*	Parameter 3 passed to extended SVC handler

Explanation

This system call calls the extended SVC handler having the extended function code specified by *s_fncd*.

Remark When this system call is used to call an extended SVC handler, the interface library for the extended SVC handlers need not be coded.

Return value

*E_OK	0	Normal termination
*E_NOSPT	–17	The viss_svc system call is not defined as CF, or this system call calls an extended SVC handler that is not registered.
E_PAR	–33	Invalid specification of extended function code (*s_fncd* ≤ 0, maximum number of extended SVC handlers that can be registered < *s_fncd*)
Others		Return value from extended SVC handler

APPENDIX A PROGRAMMING METHODS

This appendix explains how to describe processing programs when using the NEC Electronics V850 Series C compiler CA850 or the C cross V800 compiler CCV850 manufactured by Green Hills Software, Inc.

A.1 Overview

In the RX850 Pro, processing programs are classified according to purpose, as shown below.

- Task
  The minimum unit of a processing program that can be executed by the RX850 Pro.

- Directly activated interrupt handler
  A routine dedicated to interrupt processing. When an interrupt occurs, this handler is activated without using the RX850 Pro. Accordingly, a high-speed response as close as the hardware limitation can be expected.

- Indirectly activated interrupt handler
  A routine dedicated to interrupt processing. When an interrupt occurs, this handler is activated upon the completion of the interrupt preprocessing by the RX850 Pro (such as saving the contents of the registers or switching the stack).
  Compared with the directly activated interrupt handler, the response is slower. However, this handler has the advantage of simplicity for the processing within the handler because of the interrupt preprocessing by the RX850 Pro.

- Cyclically activated handler
  A routine dedicated to cyclic processing. Every time the specified time elapses, this handler is activated immediately. This routine is handled independently of tasks. When the activation time has been reached, 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, and control is passed to the cyclically activated handler.
  A cyclically activated handler incurs a smaller overhead before the start of execution, relative to any other cyclic processing programs written by the user.

- Extended SVC handler
  A function registered by the user as an extended system call.

These processing programs have their own basic formats according to the general conventions or conventions to be applied when the RX850 Pro is used.

A.2 Keywords

The character strings listed below are reserved as keywords for the configurator. These strings cannot, therefore, be used for other purposes.

clkhdr	clktim	cyc	defstk	flg
flgsvc	ini	inthdr	intstk	intsvc
maxcyc	maxflg	maxint	maxintfactor	maxmbx
maxmpl	maxpri	maxsem	maxsvc	maxtsk
mbx	mbxsvc	mem	mpl	mplsvc
no_use	prtflg	prtmbx	prtmpl	prtsem
prttsk	RX850PRO	rxsers	sct_def	sem
semsvc	ser_def	sit_def	SPOL0	SPOL1
svc	syssvc	TA_ASM	TA_DISINT	TA_ENAINT
TA_HLNG	TA_MFIFO	TA_MPRI	TA_TFIFO	TA_TPRI
TA_WMUL	TA_WSGL	TCY_OFF	TCY_ON	timsvc
tsk	tsksvc	TTS_DMT	TTS_RDY	UPOL0
UPOL1				

A.3 Reserved Words

The character strings listed below are reserved as external symbols for the RX850 Pro. These strings cannot, therefore, be used for other purposes.

_x_	_f_	_e_	_rx_

Remark The use of these character strings is prohibited when a single load module is created. There is no problem if a symbol starting with any of these character strings is used when a load module that separates the RX850 Pro and application is created.

A.4 Tasks

A.4.1 CA850-supported version

When describing a task in C language, describe it as a void-type function having one INT-type argument after function declaration by pragma directive.

The activation code specified when the sta_tsk system call is issued is set for the argument (*stacd*).

Figure A-1 shows the task description format (in C language) when the CA850 is used.

Figure A-1. Task Description Format When Using CA850 (C Language)

```
#include  <stdrx85p.h>

#pragma  rtos_task  func_task
void
func_task(INT stacd)
{
          /* Processing of task func_task */
           ..........................
           ..........................
           ..........................

          /* Termination of task func_task */
          ext_tsk();
}
```

Remark For details about the function declaration by pragma directive, refer to the **CA850 C Language User's Manual (U16054E)**.

When describing a task in assembly language, describe it as a function conforming to the function call conventions of the CA850.

The activation code specified when the sta_tsk system call is issued is set for the argument (r6 register).

Figure A-2 shows the task description format (in assembly language) when the CA850 is used.

Figure A-2. Task Description Format When Using CA850 (Assembly Language)

```
.include    "stdrx85p.inc"

            .text
            .align        4
            .globl        _func_task
_func_task:
            # Processing of task func_task
             ........................
             ........................
             ........................

            # Termination of task func_task
            jr            _ext_tsk
```

A.4.2 CCV850-supported version

When describing a task in C language, describe it as a void-type function having one INT-type argument. The activation code specified upon the issuance of the sta_tsk system call is set for the argument (*stacd*). Figure A-3 shows the task description format (in C language) when the CCV850 is used.

Figure A-3. Task Description Format When Using CCV850 (C Language)

```
#include   <stdrx85p.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.

The activation code specified upon the issuance of the sta_tsk system call is set for the argument (r6 register).

Figure A-4 shows the task description format (in assembly language) when CCV850 is used.

Figure A-4. Task Description Format When Using CCV850 (Assembly Language)

```
#include   <stdrx85p.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 compiling a task in assembly language, specify "-D_ _asm_ _" as the option at compilation.

A.5 Directly Activated Interrupt Handler

A.5.1 CA850-supported version

Use assembly language when describing a directly activated interrupt handler. However, the processing itself can be described in C language and called by the jarl instruction.

The registers must be saved before the directly activated interrupt handler processing and restored after the processing. However, the RX850 Pro provides a macro that describes the process of saving and restoring the registers in order to mitigate the workload of the user in describing a handler in assembly language.

The following figure shows the description format (in assembly language) of the directly activated interrupt handler when the CA850 is used.

Figure A-5. Description Format of Directly Activated Interrupt Handler When Using CA850 (Assembly Language)

```
.include   "stdrx85p.inc"
           .section   "int_name", text
           jr         _func_inthdr

           .text
           .align     4
           .globl     _func_inthdr
_func_inthdr:
           /* Saving registers, switching stack */
           RTOS_IntEntry

           /* Main processing of directly activated interrupt handler */
           .extern    _inthdr_body
           jarl       _inthdr_body, lp

           /* r10: ID of task to be woken up after returning from handler */
           /* Switching stack, restoring registers */
           /* Return from directly activated interrupt handler and waking up task */
           RTOS_IntReturnWakeup r10
```

```
#include   <stdrx85p.h>

ID
inthdr_body( )
{

          __asm ("mov #__tp_TEXT, tp");
          __asm ("mov #__gp_DATA, gp");
          /* Processing of directly activated interrupt handler func_inthdr */
           .....................................................
           .....................................................
           .....................................................
          /* Return from directly activated interrupt handler func_inthdr */
          return     tskid
}
```

First, describe the interrupt handler entry processing (jr instruction) at the handler address. Refer to the second and third rows in this example.

Next, describe the interrupt handler main unit processing.

The macro RTOS_IntEntry notifies the RX850 Pro of the activation of the handler, the saving of the temporary register and lp, and the switching of the task. The other registers (r20 to r30) are then saved, and control is transferred to the handler. In the above example, the C function, *inthdr_body*, of the handler is called. Before the execution of the handler main unit processing, set the TP (text pointer) and GP (global pointer) used by the handler. As described in **5.3 Directly Activated Interrupt Handler**, the values of the GP and TP become undefined. Since this setting must be described in assembly language, use the _ _asm instruction as in the above example or the #pragma asm to pragma endasm directives to describe the handler in C language. In the handler, “the system calls that can be issued from the handler” explained in the user’s manual can be issued.

When the issuance processing of the handler is completed, the registers saved by the user must be restored and execution must return from the interrupt handler. To wake up a task specified after execution has returned from an interrupt, the ID of the task to be woken up must be set to register r10. In the above example, a task ID is returned as a return value when execution returns from *inthdr_body*, and its value is copied to r10. This operation is performed with the code output from the CA850.

To return from the handler and wake up other tasks, describe the macro RTOS_IntReturnWakeup r10. To simply return from the handler, use the macro RTOS_IntReturn. The handler can be also terminated by using the system call jr_ret_int or jr_ret_wup. In this case, however, the registers that have been saved must be restored before the system call is issued. It is easier to return from the handler by using the macros.

To return from the handler after performing only simple processing, the reti instruction can be also used. In this case, the registers must be restored before issuing the instruction. The macro RTOS_IntExit is equivalent to this instruction and restores the registers. When returning from the handler by using these instructions and macros, however, do not issue a system call in the handler. If a system call is used in the handler, use RTOS_IntReturnWakeup or RTOS_IntReturn to return from the handler.

To enable multiple interrupts during the servicing of the handler, execute ei/di after the processing of RTOS_IntEntry and before RTOS_IntReturnWakeup, RTOS_IntReturn, or RTOS_IntExit.

Remark Set a branch instruction that branches to the directly activated interrupt handler at the handler address to which the processor transfers control if an interrupt occurs. This is done by the **.section** quasi directive in Figure A-5.

For details of the .section quasi directive, refer to the **CA850 C Compiler Package Assembly Language User's Manual (U16042E)**. Specify an interrupt request name defined in the device file as "*int_name*".

A.5.2 CCV850-supported version

Use assembly language when describing a directly activated interrupt handler. However, the processing itself can be described in C language and called by the jarl instruction.

The registers must be saved before the directly activated interrupt handler processing and restored after the processing. However, the RX850 Pro provides a macro that describes the process of saving and restoring the registers in order to mitigate the workload of the user in describing a handler in an assembly language.

The following figure shows the description format (in assembly language) of the directly activated interrupt handler when the CCV850 is used.

Figure A-6. Description Format of Directly Activated Interrupt Handler When Using CCV850 (Assembly Language)

```
.include   "stdrx85p.inc"
           .org       handler_address_number
           jr         _func_inthdr

           .text
           .align     4
           .globl     _func_inthdr
_func_inthdr:
           /* Saving registers, switching stack */
           RTOS_IntEntry

           /* Main processing of directly activated interrupt handler */
           .extern    _inthdr_body
           jarl       _inthdr_body, lp

           /* r10: ID of task to be woken up after returning from handler */
           /* Switching stack, restoring registers */
           /* Return from directly activated interrupt handler and waking up task */
            jr        RTOS_IntReturnWakeup r10
```

```
#include   <stdrx85p.h>

ID
inthdr_body( )
{

          __asm ("__tp, tp");
          __asm ("__gp, gp");
          /* Processing of directly activated interrupt handler func_inthdr */
           ..........................................................
           ..........................................................
           ..........................................................
          /* Return from directly activated interrupt handler func_inthdr */
          return     tskid
}
```

First, describe the interrupt handler entry processing (jr instruction) at the handler address. Refer to the second and third rows in this example.

Next, describe the interrupt handler main unit processing.

The macro RTOS_IntEntry notifies the RX850 Pro of the activation of the handler, the saving of the temporary register and lp, and the switching of the task. The other registers (r20 to r30) are then saved, and control is transferred to the handler. In the above example, the C function, *inthdr_body*, of the handler is called. Before the execution of the handler main unit processing, set the TP (text pointer) and GP (global pointer) used by the handler. As described in **5.3 Directly Activated Interrupt Handler**, the values of the GP and TP become undefined. Since this setting must be described in assembly language, use the _ _asm instruction as in the above example or the #pragma asm to pragma endasm directives to describe the handler in C language. In the handler, “the system calls that can be issued from the handler” explained in the user’s manual can be issued.

When the issuance processing of the handler is completed, the registers saved by the user must be restored and execution must return from the interrupt handler. To wake up a task specified after execution has returned from an interrupt, the ID of the task to be woken up must be set to register r10. In the above example, a task ID is returned as a return value when execution returns from *inthdr_body*, and its value is copied to r10. This operation is performed with the code output from the CCV850.

To return from the handler and wake up other tasks, describe the macro RTOS_IntReturnWakeup r10. To simply return from the handler, use the macro RTOS_IntReturn. The handler can be also terminated by using the system call jr_ret_int or jr_ret_wup. In this case, however, the registers that have been saved must be restored before the system call is issued. It is easier to return from the handler by using the macros.

To return from the handler after performing only simple processing, the reti instruction can be also used. In this case, the registers must be restored before issuing the instruction. The macro RTOS_IntExit is equivalent to this instruction and restores the registers. When returning from the handler by using these instructions and macros, however, do not issue a system call in the handler. If a system call is used in the handler, use RTOS_IntReturnWakeup or RTOS_IntReturn to return from the handler.

To enable multiple interrupts during the servicing of the handler, execute ei/di after the processing of RTOS_IntEntry and before RTOS_IntReturnWakeup, RTOS_IntReturn, or RTOS_IntExit.

Remark Set a branch instruction that branches to the directly activated interrupt handler at the handler address to which the processor transfers control if an interrupt occurs. This is done by the **.org** instruction in Figure A-6.

Specify the handler address of an interrupt as *handler_address_number*.

Caution When describing a directly activated interrupt handler in assembly language, specify ".850" as the file extension.

A.6 Indirectly Activated Interrupt Handler

A.6.1 CA850-supported version

When describing an indirectly activated interrupt handler in C language, describe it as an ID-type function having no argument.

Figure A-7 shows the description format of an indirectly activated interrupt handler (in C language) when the CA850 is used.

Figure A-7. Description Format of Indirectly Activated Interrupt Handler When Using CA850 (C Language)

```
#include   <stdrx85p.h>

ID
func_inthdr()
{
            /* Processing of indirectly activated interrupt handler func_inthdr */
             ..........................................................
             ..........................................................
             ..........................................................

            /* Return processing from indirectly activated interrupt handler func_inthdr */
            return(TSK_NULL);
}
```

Remark An indirectly activated interrupt handler is a subroutine called by interrupt processing in the nucleus. Therefore, when an indirectly activated interrupt handler is described, an instruction for branching to the indirectly activated interrupt handler needs to be set for the handler address to which the processor passes control upon the occurrence of an interrupt. This setting must be described in assembly language.

However, because the RX850 Pro provides the processing that should be described as the branch instruction in the form of a macro, this macro should be used. For example, to use the INTP100 (address: 0x100) maskable interrupt as an indirectly activated interrupt handler, describe as follows.

```
.section "INTP100"
RTOS_IntEntry_Indirect
```

The same description is required for timer interrupts since they are handled as indirectly activated interrupt handlers.

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-8 shows the description format of an indirectly activated interrupt handler (in assembly language) when the CA850 is used.

Figure A-8. Description Format of Indirectly Activated Interrupt Handler When Using CA850 (Assembly Language)

```
.include   "stdrx85p.inc"

           .text
           .align       4
           .globl       _func_inthdr
_func_inthdr:
           # Processing of indirectly activated interrupt handler func_inthdr
            ..........................................................
            ..........................................................
            ..........................................................

           # Return processing from indirectly activated interrupt handler func_inthdr
           mov          TSK_NULL, r10
           jmp          [lp]
```

Remark An indirectly activated interrupt handler is a subroutine called by interrupt processing in the nucleus. Therefore, when an indirectly activated interrupt handler is described, an instruction for branching to the indirectly activated interrupt handler needs to be set for the handler address to which the processor passes control upon the occurrence of an interrupt. This setting must be described in assembly language.

However, because the RX850 Pro provides the processing that should be described as the branch instruction in the form of a macro, this macro should be used. For example, to use the INTP100 (address: 0x100) maskable interrupt as an indirectly activated interrupt handler, describe as follows.

```
.section "INTP100"
RTOS_IntEntry_Indirect
```

The same description is required for timer interrupts since they are handled as indirectly activated interrupt handlers.

A.6.2 CCV850-supported version

When describing an indirectly activated interrupt handler in C language, describe it as an ID-type function having no argument.

Figure A-9 shows the description format of an indirectly activated interrupt handler (in C language) when the CCV850 is used.

Figure A-9. Description Format of Indirectly Activated Interrupt Handler When Using CCV850 (C Language)

```
#include <stdrx85p.h>

ID
func_inthdr()
{
        /* Processing of indirectly activated interrupt handler func_inthdr */
         ..........
         ..........
         ..........

        /* Return processing from indirectly activated interrupt handler func_inthdr */
        return(TSK_NULL);
}
```

Remark An indirectly activated interrupt handler is a subroutine called by interrupt processing in the nucleus. Therefore, when an indirectly activated interrupt handler is described, an instruction for branching to the indirectly activated interrupt handler needs to be set for the handler address to which the processor passes control upon the occurrence of an interrupt. This setting must be described in assembly language.

However, because the RX850 Pro provides the processing that should be described as the branch instruction in the form of a macro, this macro should be used. For example, to use the INTP100 (address: 0x100) maskable interrupt as an indirectly activated interrupt handler, describe as follows.

```
.org 00000100
RTOS_IntEntry_Indirect
```

The same description is required for timer interrupts since they are handled as indirectly activated interrupt handlers.

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-10 shows the description format of an indirectly activated interrupt handler (in assembly language) when the CCV850 is used.

Figure A-10. Description Format of Indirectly Activated Interrupt Handler When Using CCV850 (Assembly Language)

```
#include   <stdrx85p.h>

           .text
           .align     4
           .globl     _func_inthdr
_func_inthdr:
           # Processing of indirectly activated interrupt handler func_inthdr
            ..........................................................
            ..........................................................
            ..........................................................

           # Return processing from indirectly activated interrupt handler func_inthdr
           mov        TSK_NULL, r10
           jmp        [lp]
```

Remark An indirectly activated interrupt handler is a subroutine called by interrupt processing in the nucleus. Therefore, when an indirectly activated interrupt handler is described, an instruction for branching to the indirectly activated interrupt handler needs to be set for the handler address to which the processor passes control upon the occurrence of an interrupt. This setting must be described in assembly language.

However, because the RX850 Pro provides the processing that should be described as the branch instruction in the form of a macro, this macro should be used. For example, to use the INTP100 (address: 0x100) maskable interrupt as an indirectly activated interrupt handler, describe as follows.

```
.org 00000100
RTOS_IntEntry_Indirect
```

The same description is required for timer interrupts since they are handled as indirectly activated interrupt handlers.

Caution When describing an indirectly activated interrupt handler in assembly language, specify ".850" as the file extension.

A.7 Cyclically Activated Handler

A.7.1 CA850-supported version

When describing a cyclically activated handler in C language, describe it as a void-type function having no argument.

Figure A-11 shows the description format of a cyclically activated handler (in C language) when the CA850 is used.

Figure A-11. Description Format of Cyclically Activated Handler When Using CA850 (C Language)

```
#include   <stdrx85p.h>

void
func_cychdr()
{
            /* Processing of cyclically activated handler func_cychdr */
            ..........................................................
            ..........................................................
            ..........................................................

            /* Return processing from cyclically activated handler func_cychdr */
            return;
}
```

Remark A cyclically activated handler is a subroutine called by system clock processing in the nucleus.

When describing a cyclically activated handler in assembly language, describe it as a function conforming to the function call conventions of the CA850.

Figure A-12 shows the description format of a cyclically activated handler (in assembly language) when the CA850 is used.

Figure A-12. Description Format of Cyclically Activated Handler When Using CA850 (Assembly Language)

```
.include   "stdrx85p.inc"

           .text
           .align       4
           .globl       _func_cychdr
_func_cychdr:
           # Processing of cyclically activated handler func_cychdr
           ..................................................
           ..................................................
           ..................................................

           # Return processing from cyclically activated handler func_cychdr
           jmp          [lp]
```

Remark A cyclically activated handler is a subroutine called by system clock processing in the nucleus.

A.7.2 CCV850-supported version

When describing a cyclically activated handler in C language, describe it as a void-type function having no argument.

Figure A-13 shows the description format of a cyclically activated handler (in C language) when the CCV850 is used.

Figure A-13. Description Format of Cyclically Activated Handler When Using CCV850 (C Language)

```
#include   <stdrx85p.h>

void
func_cychdr()
{
            /* Processing of cyclically activated handler func_cychdr */
             .....................................................
             .....................................................
             .....................................................

            /* Return processing from cyclically activated handler func_cychdr */
            return;
}
```

Remark A cyclically activated handler is a subroutine called by system clock processing in the nucleus.

When describing a cyclically activated handler in assembly language, describe it as a function conforming to the function call conventions of the CCV850.

Figure A-14 shows the description format of a cyclically activated handler (in assembly language) when the CCV850 is used.

Figure A-14. Description Format of Cyclically Activated Handler When Using CCV850 (Assembly Language)

```
#include   <stdrx85p.h>

           .text
           .align     4
           .globl     _func_cychdr
_func_cychdr:
           # Processing of cyclically activated handler func_cychdr
            ..................................................
            ..................................................
            ..................................................

           # Return processing from cyclically activated handler func_cychdr
           jmp        [lp]
```

Remark A cyclically activated handler is a subroutine called by system clock processing in the nucleus.

Caution When describing a cyclically activated handler in assembly language, specify ".850" as the file extension.

A.8 Extended SVC Handler

A.8.1 CA850-supported version

When describing an extended SVC handler in C language, describe it as an INT-type function.

Figure A-15 shows the description format of an extended SVC handler (in C language) when the CA850 is used.

Figure A-15. Description Format of Extended SVC Handler When Using CA850 (C Language)

```
#include   <stdrx85p.h>

INT
func_svchdr(VW prm1, VW prm2, VW prm3)
{
           int      ret;

           /* Processing of extended SVC handler func_svchdr */
           ..................................................
           ..................................................
           ..................................................

           /* Return processing from extended SVC handler func_svchdr */
           return(INT ret);
}
```

When describing an extended SVC handler in assembly language, describe it as a function conforming to the function call conventions of the CA850.

Figure A-16 shows the description format of an extended SVC handler (in assembly language) when the CA850 is used.

Figure A-16. Description Format of Extended SVC Handler When Using CA850 (Assembly Language)

```
.include   "stdrx85p.inc"

           .text
           .align       4
           .globl       _func_svchdr
_func_svchdr:
           # Processing of extended SVC handler func_svchdr
           ..................................................
           ..................................................
           ..................................................

           # Return processing from extended SVC handler func_svchdr
           mov          ret, r10
           jmp          [lp]
```

A.8.2 CCV850-supported version

When describing an extended SVC handler in C language, describe it as an INT-type function.

Figure A-17 shows the description format of an extended SVC handler (in C language) when the CCV850 is used.

Figure A-17. Description Format of Extended SVC Handler When Using CCV850 (C Language)

```
#include   <stdrx85p.h>

INT
func_svchdr(VW prm1, VW prm2, VW prm3)
{
           int       ret;

           /* Processing of extended SVC handler func_svchdr */
            ..................................................
            ..................................................
            ..................................................

           /* Return processing from extended SVC handler func_svchdr */
           return(INT ret);
}
```

When describing an extended SVC handler in assembly language, describe it as a function conforming to the function call conventions of the CCV850.

Figure A-18 shows the description format of an extended SVC handler (in assembly language) when CCV850 is used.

Figure A-18. Description Format of Extended SVC Handler When Using CCV850 (Assembly Language)

```
#include   <stdrx85p.h>

           .text
           .align     4
           .globl     _func_svchdr
_func_svchdr:
           # Processing of extended SVC handler func_svchdr
           ................................................
           ................................................
           ................................................

           # Return processing from extended SVC handler func_svchdr
           mov        ret, r10
           jmp        [lp]
```

Caution When describing an extended SVC handler in assembly language, specify ".850" as the file extension.

APPENDIX B Q & A

[Product Overview]

Q.1

Is the RX850 Pro compliant with either the ITRON or μITRON specification?

A.1

The RX850 Pro is compliant with the μITRON3 specification.

Q.2

What is the RX850 Pro's nucleus (kernel) size?

A.2

It ranges from 5 to 13 KB.

This varies depending on the number of system calls used.

When all system calls are used, the size is 13 KB.

For details of size estimates, see the "RX850 Pro Memory Size Estimate page" in the FAQ at the NEC Electronics Microcomputer website.

Q.3

What RAM size does the RX850 Pro use?

A.3

The RAM size used by the RX850 Pro depends on the application.

The number of control blocks varies depending on the number of created tasks and other resources.

For description of control block sizes and methods for estimating the RAM sizes used, see **CHAPTER 5 MEMORY AND MEMORY CAPACITY ESTIMATION** in the **RX850 Pro Installation User's Manual (U13774E)** or use the "RX850 Pro Memory Size Estimate page" in the FAQ at the NEC Electronics Microcomputer website.

Q.4

What language is the RX850 Pro coded in?

A.4

It is coded entirely in assembly language.

Q.5

How is the RX850 Pro incorporated into applications?

A.5

The RX850 Pro provides libraries (librxp.a/librxpm.a) containing system call processing blocks and also provides common processing blocks as objects (rxcore.o/rxtmcore.o).

In other words, the RX850 Pro is incorporated into applications by system call processing blocks that reference libraries when linked and common processing blocks that link to objects.

See Q.65 for description of differences between librxp.a and librxpm.a, and Q.111 for description of differences between rxcore.o and rxtmcore.o.

Q.6

When should the RX850 be used and when should the RX850 Pro be used?

A.6

Although both the RX850 and the RX850 Pro comply with the μITRON3 specification, they differ in some ways. Their main differences are described below.

- The RX850 enables only static generation of resources while the RX850 Pro enables both static and active generation.
- The RX850's memory pools include both variable-length and fixed-length memory pools while the RX850 Pro has only variable-length memory pools.

The RX850 is positioned as a subset of the RX850 Pro, with a smaller code size and less RAM capacity usage than the RX850 Pro.

The RX850 is also slightly faster than the RX850 Pro.

However, since it uses r0 relative instructions, certain restrictions apply as to how memory is allocated.

By contrast, the RX850 Pro has a larger code size and more RAM capacity usage than the RX850, and no such restrictions apply to how memory is allocated.

Also, since it enables resources to be generated actively, it allows more leeway in creating applications.

Either RX version is able to operate in V850 Series devices, but, in the view of the differences described above, we recommend the RX850 when using V850 Series devices with smaller memory resources and the RX850 Pro when using V850 Series devices with more memory and higher MIPS values.

It may be useful to refer to a document for more detailed specification of these differences (RX850/RX850 Pro specification comparison, document number: SUD-T-4961). To obtain this document, contact an NEC Electronics sales representative.

Q.7

Which compilers does the RX850 Pro support?

A.7

Currently, the RX850 Pro supports the NEC Electronics CA850 compiler and the GHS (Green Hills Software) CCV850E compiler.

The RX850 Pro product is packaged with support for both of these compilers, and one or the other must be selected at the time of installation.

Q.8

I'm interested in using the RD850 Pro task debugger. Is that sold separately?

A.8

The RD850 Pro is standard-equipped with the RX850 Pro.

For use instructions, see the **RD850 Pro User's Manual**.

Q.9

I'm interested in using the AZ850 system performance analyzer. Is that sold separately?

A.9

The AZ850 is not equipped with the RX850 Pro (it is sold separately). However, it is included in the SP850 package.

The AZ850 can be used to obtain information such as task transitions and CPU usage rates.

Q.10

What caution points should I note when migrating from RX850 Ver. 3.13 to RX850 Pro?

A.10

At the system call level, compatibility is maintained, so there is no need to modify the programs per se. However, the argument types used in system calls differ slightly.

The layout and location of links in system information, etc., and the memory management methods differ significantly, and so those aspects must be reviewed and modified accordingly.

The migration procedures are described in a document (document excerpted from the RX850 Pro RX850 (Ver. 3.1x), document number: SUD-T-4816), so refer to this document. To obtain this document, contact an NEC Electronics sales representative.

Q.11

What is a CF (configuration) definition file?

A.11

A CF definition file is a file that contains various data that is provided to the nucleus (kernel).

The user describes various information (task information, etc.) using a specified format, and the file is then processed by a configurator (cf850pro) and converted to an assembly language file.

Q.12

What is defined by the RX850 Pro's CF definition files?

A.12

The configuration files define system information (SIT) and system call information (SCT).

The SIT (System Information Table) information includes the following:

- System information
- System maximum value information
- System memory information
- Task information
- Event flag information
- Semaphore information
- Mailbox information
- Memory pool information
- Cyclic handler information
- Priority information
- Indirectly activated interrupt handler information
- Clock interrupt information
- Extended SVC handler information

The SCT (System Call Table) information includes the following:

- Task management/task-associated synchronization management function system call information
- Synchronous communication management function system call information
- Memory pool management function system call information
- Time management function system call information
- System management function system call information

Q.13

How are CF definition files created?

A.13

They are text files that are created using a specified format.

For description of the format, see **CHAPTER 6 CONFIGURATION FILE** in the **RX850 Pro Installation User's Manual (U13774E)**.

Q.14

What is the configurator (cf850pro)?

A.14

The configurator is an application that uses user-generated information from CF definition files to create table information and other information needed by the real-time OS (RX850 Pro).

The configurator is an MS-DOS application and the executable file (for the RX850 Pro) is called cf850pro.exe.

The configurator can be started by entering a command at the DOS prompt.

It can also be started from NEC Electronics PM plus (Ver. 3.15 or later).

In other words, when the configuration file is updated, cf850pro is automatically started in order to create SIT and SCT information.

Q.15

What are the options when starting the configurator (cf850pro)?

A.15

The options when starting the configurator are as follows.

cf850pro -i *sit_file -c sct_file -d h_file cf_file* [return]

sit_file: System information table (.sit)
sct_file: System call table (.sct)
h_file: System information header file (.h)
cf_file: CF definition file (.cf)

Q.16

What is the “key ID” that is specified when used to create resources in CF definition files?

A.16

A key ID is required if the setting “automatically assign ID numbers” is entered when creating resources.

Users do not need to pay attention to key IDs when manually assigning ID numbers.

Specify “0” to enter the setting “do not use key ID.”

When auto creation of ID numbers has been specified, ID numbers are automatically assigned by the configurator and the user is not able to tell what the ID numbers of resources are.

That is why key IDs are needed.

A unique key ID is assigned to each resource when the resource is created.

Once the user has the specified key ID number, the user can use a system call (vget_*id) to acquire the ID number of the corresponding resource.

Q.17

When the following is set and executed in a CF definition file:

```
mem        SPOL0        0xffffd200      0x0000fe0
```

the configurator (CF850 Pro) outputs the following error message.

E2259: enough system memorypool “SPOL0” block size

The memory estimates are described below.
Although the memory pools are not used, what areas must be allocated other than for the task stacks, interrupt stacks, and management objects?

Task A stack		0x1f4 bytes
Task B stack		0x3e8 bytes
Task C stack		0x3e8 bytes
Sub total		0x9c4 bytes
Interrupt stack		0x1f4 bytes
Management objects		
System management table	504 + 4 + 16 + 32	= 556 bytes
Task management block	3 × 56	= 168 bytes
Semaphore management block	1 × 20	= 20 bytes
Interrupt handler	6 × 16 + 104	= 200 bytes
Cyclically activated time management	3 × 40	= 120 bytes
Sub total		1064 bytes (0x428)
Total	0x9c4 + 0x1f4 + 0x428 = 0xfe0	

A.17

The task stack size requires that 100 bytes (28 bytes for the task stack management table and 72 bytes for the context area) be added to the size specified by the CF definition file.

For details, see **5.3 Capacity of Task Stack** in the **RX850 Pro Installation User’s Manual (U13774E)** or use the “RX850 Pro Memory Size Estimate page” in the FAQ at the NEC Electronics Microcomputer website.

Task A stack	0x1f4 + 100	= 600 bytes
Task B stack	0x3e8 + 100	= 1100 bytes
Task C stack	0x3e8 + 100	= 1100 bytes
Sub total		2800 bytes

In addition, the interrupt handler stack frame size (idle task = 80 bytes) must be added to the interrupt handler stack area.

For details, see **5.4 Capacity of Stack for Interrupt Handler** in the **RX850 Pro Installation User's Manual (U13774E)**.

Interrupt stack	0x1f4 + 80	= 580 bytes
Total	2800 + 580 + 1064	= 4444 bytes (0x115c)

In such cases, a value of 0x115c or more must be specified to SPOL0 in the CF definition file.

Q.18

How are tasks created?

A.18

They are created statically by CF definition files or when the "cre_tsk system call" is issued from an application.

Settings that are entered when a task is created include the task name (ID), task activation address, task stack size, task initial priority, task initial status, the task activation code, interrupt status at activation, values specific to the GP and TP registers, and the key ID number.

Q.19

How many tasks can be created?

A.19

The number of tasks specified by the maxtsk (maximum number of tasks) value can be created.

However, the highest value that can be specified for maxtsk as this maximum number is 32767, so up to 32767 tasks can be created.

Q.20

What is the initialization handler?

A.20

The initialization handler is the "handler that is executed before the scheduler is activated." The scheduler is activated as part of the RX850 Pro's processing that occurs after a startup routine activates the RX850 Pro's initialization processing.

The initialization handler's address can be specified by the CF definition file (it is specified by the "ini" setting).

In the sample, this is the address called "varfunc."

Thus, the initialization handler provides a convenient means for coding processing that precedes system activation, such as task activation and hardware initialization.

In cases where use of the initialization handler is not desired, it should not be specified in the CF definition file.

If this specification is omitted, the RX850 Pro will operate without an initialization handler, and will therefore have a slightly smaller code size.

Q.21

Does the RX850 Pro include any idle tasks?

A.21

The default setting is that it does not include idle tasks, but the user is able to create tasks that have the lowest priority in the system and use them as idle tasks.

In cases where there are neither idle tasks nor other tasks to be executed, the OS issues a HALT instruction to stop the CPU.

An idle task must be created in order to use the “STOP” and “IDLE” modes that are available as low power modes in the V850 Series devices.

In other words, this can be implemented by writing code that sets “STOP” or “IDLE” into an idle task. When this is done, the idle task must be created as the lowest-priority task in the system.

Q.22

What happens when there are no more tasks to be executed?

A.22

The CPU enters HALT mode.

In other words, HALT processing is executed in the RX850 Pro.

Q.23

What is the task priority range in the RX850 Pro?

A.23

This range must be within the “task priority range (maxpri)” specified in the CF definition file.

This priority specification range is from 1 to 252.

Q.24

Are there any caution points to note when specifying the task priority range?

A.24

Only set a range within the task priority range that will actually be used.

The reason for doing this is that it affects the amount of time the RX850 Pro needs to search the ready queue.

Only the range of “task priority numbers” set in the CF definition file will be used to create the ready queue (which requires four bytes per priority number).

In other words, a larger range of task priority numbers increases the RAM size.

In addition, when searching for the task used to activate the RX850 Pro scheduler, problems may occur if the target task is not found quickly and, in the worst case, all numbers from the highest to lowest priority may have to be searched.

This results in longer processing times.

That is why it is best to restrict the priority numbers to the range actually needed for use.

Q.25

How is a task stack reserved in memory?

A.25

Only the amount of memory set by the “task stack size” specified in the CF definition file’s task information is reserved from the stack area.

The stack area can be specified in either the SPOL0 or SPOL1 system memory area.

These addresses are set by the “system memory information (mem)” specified in the CF definition file.

Note that no problems occur when the task stack is set only to the SPOL0 or only to the SPOL1 area.

Q.26

How large should be task stack be?

A.26

The size of the task stack depends on the application.

The stack is used for the following purposes.

- To save the task context when a task is preempted
- To save register information when an interrupt occurs
- To save local variables

The task context size and stack frame size during interrupts are clearly specified.

For details of these, see **CHAPTER 5 MEMORY AND MEMORY CAPACITY ESTIMATION** in the **RX850 Pro Installation User’s Manual**.

The local variables also differ, depending on how many are used and their sizes.

When using an array that has a large number of elements, a fairly large stack is utilized.

During development, or at other times stack size estimates are only vague, a large stack should be reserved. The stack size can then be reduced if possible later, after a clearer estimate has been made.

Also, another means of estimating is to use the RD850 Pro to compare the initial task’s stack pointer value (SP value) with the stack pointer value when the application is operating (i.e., when switching to another task).

Q.27

When several tasks are being created, are task stacks allocated separately to internal RAM and external RAM?

A.27

Yes, they are.

Task stack areas are defined in CF definition files at the same time as the tasks themselves.

Although task stacks are allocated to one of the system memory pools (SPOL0 or SPOL1), and when SPOL0 and SPOL1 are defined, they should be allocated either to internal RAM or external RAM.

When a task is created statically in a CF definition file is given as an example.

For example, when SPOL0 is allocated to external RAM and SPOL1 to internal RAM, SPOL0 and SPOL1 are defined as follows.

```
mem SPOL0 0x200000 0x1000
mem SPOL1 0xffffe000 0x1000
```

Furthermore, when TASK01 stack (0x200 bytes) is allocated to external RAM and TASK02 stack (0x100 bytes) is allocated to internal RAM, these tasks are defined as follows.

```
tsk TASK01 TTS_RDY 0x0 0x0 TA_ASM _task01 0x8 TA_ENAINT ¥
0x200:SPOL1 no_use no_use 0x0

tsk TASK02 TTS_RDY 0x0 0x0 TA_ASM _task02 0x6 TA_ENAINT ¥
0x100:SPOL0 no_use no_use 0x0
```

Also, when tasks are created actively by the cre_tsk system call, they are specified by the arguments in cre_tsk.

Q.28

When the rel_wai system call (forced wake-up wait release) is issued to forcibly release the wait status of other tasks, are the target number of task wake-up requests cleared?

A.28

The rel_wai system call does not clear the number of task wake-up requests.

In other words, when there have been multiple wake-up requests, the request count remains the same.

The can_wup system call is used to clear the number of wake-up requests to 0.

Q.29

Can the #pragma rtos_task instruction that is included in the NEC Electronics compiler (CA850) be used?

A.29

Yes, it can be used.

When this instruction is used, the compiler recognizes the specified function type as being for a real-time OS task and outputs code accordingly.

With an ordinary function, prologue processing and epilogue processing of the function are required and the corresponding code is output by the compiler. However, this code is not needed in the real-time OS.

When the #pragma rtos_task instruction is used to specify function type as being a task, there is no output of the code for prologue processing and epilogue processing of the function and the code size is therefore reduced.

Q.30

How is the mailbox created?

A.30

It can be created statically by a CF definition file or by issuing a cre_mbx system call from an application.

Q.31

How many mailboxes can be created?

A.31

The number of mailboxes up to the specified maximum number can be created.

However, the highest value that can be specified as the maximum number is 32767, so up to 32767 mailboxes can be created.

Q.32

Are there any restrictions on the sizes of messages when using mailboxes?

A.32

There are no such restrictions.

Any size allowed by the available memory can be used.

Q.33

Are the messages themselves being sent to mailboxes?

A.33

No, only the messages' start addresses (pointers) are being sent.

Q.34

What area should be used for writing messages?

A.34

For the sake of facilitating memory area management, we recommend using memory blocks that are managed by the RX850 Pro (see Q.54).

The specific sequence is to obtain (with get_blk) the memory area to be used for messages, write messages to that area, and then send the message (with snd_msg) to the mailbox.

The receiving side receives the message (with rcv_msg), reads its contents, and then releases the memory area (with rel_blk) that was used by the message.

However, when using del_mbx to actively delete the mailbox where messages are queued, that area can be released by the RX850 Pro if the message area is a memory block, but if it is an area not controlled by the RX850 Pro, the delete operation cannot be guaranteed.

Q.35

Can priorities be assigned to messages?

A.35

Yes, they can.

When a prioritized message is sent to the mailbox that has a message priority (as specified by the TA_MPRI attribute), the message is queued according to its priority order.

The priority number is stored in the message itself.

It is stored in a two-byte area that is five bytes from the start of the message.

When handling prioritized messages, writing to the target area needs to be kept organized.

Generally, the area for prioritized messages is written 8 bytes ahead of the start in order to maintain alignment.

Q.36

How are messages queued in mailboxes?

A.36

There are two methods for queuing messages in mailboxes.

One method is called “order of message admission (FIFO order)” and other is called “order of message priority (PRI order)”.

One of these queuing methods is specified as a mailbox attribute in a CF definition file.

In other words, the queuing method is determined separately for each mailbox.

The TA_MFIFO attribute is used to specify the first method and the TA_MPRI attribute is used to specify the second method.

Q.37

Can messages be sent for specified tasks?

A.37

There is no function that specifies the task when sending a message.

However, this sort of operation can be done by creating a dedicated mailbox to receive a particular message (see Q.30).

Q.38

If several tasks are waiting for messages, in what order are the messages transferred?

A.38

They are transferred either in order of waiting message or according to the tasks’ priority.

The order to be used is specified by an attribute for the target mailbox.

When creating a mailbox, if the specified attribute is TA_TFIFO, messages are queued in the waiting order and if the specified attribute is TA_TPRI, they are queued in order of task priority.

Reference example:

```
-- Mailbox information
mbx   0x1   0x0   TA_TFIFO TA_MFIFO              0x1
mbx   0x2   0x0   TA_TPRI  TA_MFIFO(or TA_MPRI)  0x2
```

Q.39

0xffffffab was the result when referencing the value returned by the debugger in response to the prcv_msg system call. When this value is referenced as a hexadecimal value, it is expected to be “–55” but there is no such return value of “–55” in the user’s manual. How should this return value be referenced?

A.39

The system call return value described in the user’s manual is a decimal value.

When 0xffffffab is expressed as a signed decimal number, it is “–85.”

The prcv_msg return value “–85” is returned by E_TMOUT when there are no messages in the target mailbox.

Q.40

How are event flags created?

A.40

They are created either by a CF definition file or by using the cre_flg system call in a program.

Q.41

How many event flags can be created?

A.41

The number of event flags up to the specified maximum number can be created.

However, the highest value that can be specified as the maximum number is 32767, so up to 32767 event flags can be created.

Q.42

What is size (in bits) of event flags?

A.42

32 bits

Q.43

With the RX850 Pro, can several tasks wait for an event that uses just one event flag?

A.43

Yes, they can.

When event flags are created, the setting can be either “enable only single task (TA_WSGL attribute)” or “enable multiple tasks (TA_WMUL attribute).”

Q.44

In the RX850 Ver. 3.13, there are 1-bit event flags, but are there any of these in the RX850 Pro?

A.44

No, there are not any 1-bit event flags in the RX850 Pro.

1-bit event flags are used in the RX850 Ver. 3.13, but when the program is upgraded to the RX850 Pro, all event flags must be 32-bit event flags.

Likewise, any system calls related to 1-bit event flags should be converted to 32-bit event flags.

For details of how to make these changes, see the document (SUD-T-4816-2) describing migration from the RX850 (Ver. 3.1x). To obtain this document, contact an NEC Electronics sales representative.

Q.45

What is the timing of the clear processing related to the “event flag clear specification” (TWF_CLR) for the wai_flg, twai_flg, and pol_flg arguments?

A.45

After set_flg, a separate routine is used to clear event flags; not the routine within wai_flg, twai_flg, or pol_flg.

Next, a check is done to determine whether or not any tasks are waiting for an event flag. If there are any such tasks, their wait conditions are checked, the wait status is cleared, and then the flag is cleared.

Since the contents of the separate routine operate during interrupt disable mode set by the di instruction, any set_flg specification within an interrupt will not change flag settings.

Also, with regard to task transitions, a scheduler activation request is issued following completion of processing such as clearing of wait statuses, so flag settings are not changed by other tasks.

Q.46

In cases where multiple tasks are waiting for an event set by wai_flg to occur in relation to a particular bit in a single event flag, how are tasks activated once the event has occurred?

A.46

When multiple tasks are waiting for an event set by a single event flag, if the condition for clearing the wait status is the same, once the wait status is cleared all of the waiting tasks are set to ready state.

Afterward, the scheduler is activated and the highest-priority task is set to run state.

In other words, at the time when the set_flg system call is issued, it is not just the highest-priority task that gets activated; all of the waiting tasks are activated.

After all of these tasks are activated, the activated task that has the highest priority is set to run state.

However, it is not absolutely necessary that the task that has the highest priority among the tasks that had been in event flag wait status will be set to run state.

Q.47

When the set_flg system call is issued, if the wai_flg system call was used to specify bit pattern clearing (TWF_CLR) for even one of the tasks that was waiting for the event flag set by that system call, is the flag actually cleared?

A.47

Clearing of an event flag wait state occurs after the event flag wait queue (FIFO) is checked starting from the start of the queue, when the task that establishes the wait condition is activated.

At that time, if clearing of the task has been specified, the bit pattern is cleared immediately.

Accordingly, if clearing has been specified for the task that establishes the wait condition, then the subsequent tasks that are still in the wait queue are not activated when this wait condition is established.

An example in which one task is waiting is shown in **4.4.5 Messages**.

Q.48

How are semaphores created?

A.48

They are created either by a CF definition file or by using the cre_sem system call in a program.

Q.49

How many semaphores can be created?

A.49

The number of semaphores up to the specified maximum number can be created.

However, the highest value that can be specified as the maximum number is 32767, so up to 32767 semaphores can be created.

Q.50

Where is the number of initial semaphore resources set?

A.50

It is set by a CF definition file or by the cre_sem system call when creating semaphores.

Q.51

Is there an upper limit on the number of semaphore resources?

A.51

Yes, there is an upper limit.

The upper limit is 0x7fffffff (2147483647).

Q.52

When semaphore resources exist, what happens if an operation is performed to return the resources?

A.52

Semaphores include resource counters that are incremented when such an operation is performed.

Q.53

Can management objects related to resources such as semaphores be assigned to specific resources in SPOL0 or SPOL1?

Also, if both SPOL0 and SPOL1 are defined by a CF definition file, which one is used first when they are activated?

A.53

All resources including semaphores are managed by the system.

This means that the system maintains management tables for all resources in SPOL0 and they are not assigned to SPOL1.

Before the RX850 Pro is activated, there is no concept of SPOL0 or SPOL1.

In other words, in order to use stacks, a stack area must be reserved as part of the boot processing.

See the sample programs provided with this product for further description.

After the RX850 Pro is activated, management tables are created as the system area in SPOL0.

Note that SPOL0 must be defined.

Be careful to note that it is not possible to define only SPOL1.

Q.54

How is memory management set up?

A.54

In the RX850 Pro, memory is managed in two main categories: memory pools and memory blocks.

There are several memory blocks in each memory pool.

Requests to obtain and return memory areas for these memory pools are issued by tasks or the interrupt handler.

These obtained or returned memory areas are called memory blocks.

Q.55

How are memory pools created?

A.55

They are created by a CF definition file or by issuing a cre_mpl system call.

The memory pool name (ID), task queuing method, memory pool size, and key ID are set in a CF definition file.

Q.56

How many memory pools can be created?

A.56

The number of memory pools up to the specified maximum number can be created.

However, the highest value that can be specified as the maximum number is 32767, so up to 32767 memory pools can be created.

Q.57

Are there any fixed-length memory pools?

A.57

No, all are variable-length memory pools. If fixed-length memory pools are used when migrating an application from the RX850, they must all be replaced by variable-length memory pools.

Q.58

When the system memory pools (SPOL0 and SPOL1) and/or user memory pools (UPOL0 and UPOL1) are defined by a CF definition file, is it necessary to avoid placing them in the same area as the .bss or .data section?

Also, are there any memory pools that must be placed in internal RAM?

A.58

SPOL0, SPOL1, UPOL0, and UPOL1 all specify direct addresses and they occupy an area whose size is specified starting from the specified address.

Although variables are placed in the .bss or .data section by a compiler or linker, compilers and linkers are not recognized in relation to the above-mentioned memory areas that are used by the RX850 Pro.

Consequently, layout of .bss or .data section in the SPOL0, SPOL1, UPOL0, and UPOL1 areas should be avoided.

When the linker uses link directives to reserve in advance segments to be used by the RX850 Pro, it tends to avoid problems caused by overlapping with other areas (for a relevant sample, see Q.63).

However, it is not possible to detect problems such as overflows.

Also, there are no restrictions for the RX850 Pro with regard to having to place the above memory areas in either internal or external memory space.

Q.59

In variable-length memory pools, is any “garbage collection” done to organize the unused areas that are left after repeated acquisition and release of memory blocks?

A.59

No, it is not done.

This is because the C language which is used in applications uses pointers, etc., to handle direct addresses.

If the RX850 Pro were to perform such garbage collection and move entities (memory blocks), contradictions would occur in the application between the addresses held by pointers and the actual addresses of such entities, which could cause operation faults in the application.

With the RX850 Pro, when rel_blk is used to return memory blocks, the spaces before and after each returned memory block are checked and if either (or both) of these spaces is empty (unused), it is merged to create a large empty space and minimize fragmentation of memory pool.

However, if the empty spaces are not contiguous, any memory block that is larger than the largest empty space cannot be obtained.

Q.60

What are the meanings of SPOL0, SPOL1, UPOL0, and UPOL1?

A.60

Their meanings are listed as follows.

- SPOL0 … System Memory Pool 0
- SPOL1 … System Memory Pool 1
- UPOL0 … User Memory Pool 0
- UPOL1 … User Memory Pool 1

Memory areas that are used as resources for the RX850 Pro are assigned to one of the above memory pools. Specifically:

- Interrupt stack area: SPOL0 or SPOL1
- Task stack area: SPOL0 or SPOL1
- Memory pool area: UPOL0 or UPOL1

The start address and size of SPOL0, SPOL1, UPOL0, and UPOL1 are set by CF definition files.

Setting example:

```
-- Memory information
mem     SPOL0     0x1000     0x0000
mem     SPOL1     0x2000     0x1000
mem     UPOL0     0x3000     0x7000
mem     UPOL0     0x20000    0x2500
mem     UPOL1     0x30000    0x1500
```

Q.61

Would assigned stacks only to SPOL0 (System Pool 0) cause any problems?

A.61

No, it would not cause problems.

Q.62

When a startup routine (boot.s or boot.850) is used to set a stack pointer (sp), how should the size be determined?

A.62

Before starting up the RX850 Pro, if there is any use of the stack, then the corresponding stack pointer is used. For example, when a function call is executed and the lp value at that time is retained, the target stack is the one specified by the stack pointer (sp). This is also the case when a function call occurs within a function.

In the startup routine that is included with the product as a sample, a stack is not used in the NEC Electronics version before the RX850 Pro is started up, i.e., up until the jump to kernel initialization by jmp [lp].

However, in the GHS version, there is a function call for memory initialization, so a stack is used to save the lp for that section (see Q.66).

Still, as is shown in the sample, a size as large as 0x28000 bytes is not used.

The reason why 0x28000 is specified in the sample is that the link directive file overlaps this area with the memory information used by the RX850 Pro so that, after the RX850 Pro is started, this section is used as a memory area (specified by mem).

The total size specified by mem in the sample's CF definition file is 0x28000.

In the sample, if the stack is not regulated, a user who is using a modified version of the sample might issue a function call without first setting up a stack, which can result in operation faults. That is why an ample stack size is used.

One way to determine the stack size without having to use a startup routine, is to use the debugger to check the approximate stack value and then use that value as the stack size.

Alternatively, if the size specified by mem is used, there will not be much space wasted, so that value can also be used.

Q.63

With the RX850 Pro, since the linker cannot be used to detect memory areas specified by mem in a CF definition file, there is no restriction against overlapping with memory areas used by a user program.

What should be done to detect an "overlap" error when linking?

A.63

Since the CF definition file and link directive file are not linked, the link directive file must be modified.

A method for doing this is described below, using the RX850 Pro sample program as an example.

The system memory pools and user memory pools defined by the sample CF definition file (sys.cf) are as follows.

```
-- memory information
mem SPOL0 0x00110000 0x00010000
mem SPOL1 0x00120000 0x00010000
mem UPOL0 0x00130000 0x00008000
```

This works as long as this area is reserved as a dummy section.

[NEC Electronics version]

The following code is added to the startup routine (boot.s).

```
----------------------------------------
-- /*** SPOL0 area specifying ***/
.section ".spol0",  bss
.lcomm  __spol0_head,  0x10000,  4
.lcomm  __spol0_end,   0,  4

-- /*** SPOL1 area specifying ***/
.section ".spol1",  bss
.lcomm  __spol1_head,  0x10000,  4
.lcomm  __spol1_end,   0,   4

-- /*** UPOL0 area specifying ***/
.section ".upol0",  bss
.lcomm  __upol0_head,  0x8000,    4
.lcomm  __upol0_end,   0,   4
----------------------------------------
```

Next, the following code is added to the link directive.

```
-------------------------------------------------------------
SYSPOL0  : !LOAD ?RW            V0x00110000 {
        .spol0          = $NOBITS          ?AW       .spol0;
};

SYSPOL1  : !LOAD ?RW            V0x00120000 {
        .spol1          = $NOBITS          ?AW       .spol1;
};

USERPOL0 : !LOAD ?RW            V0x00130000 {
        .upol0          = $NOBITS          ?AW       .upol0;
};
-------------------------------------------------------------
```

[NEC Electronics version]

The following code is added to the startup routine (boot.850).

```
----------------------------------------
-- /*** SPOL0 area specifying ***/
.section ".spol0",  bss

-- /*** SPOL1 area specifying ***/
.section ".spol1",  bss

-- /*** UPOL0 area specifying ***/
.section ".upol0",  bss
----------------------------------------
```

Next, the following code is added to the link directive.

```
----------------------------------------
         :
         :
.spol0  0x00110000    :
.spol1  0x00120000    :
.upol0  0x00130000    :
         :
         :
----------------------------------------
```

This enables the area to be reserved by sys.cf to be recognized by the linker.
Consequently, when the application is linked, an error can be detected if this area overlaps with another area.
However, certain other errors such as overflows cannot be detected.

Q.64

The return value for the rel_blk system call was a negative value.
A negative value should not occur. Why did this happen?

A.64

The RX850 Pro complies with the μITRON3.0 standard, and when an error occurs during μITRON3.0-compliant system call processing, the return value is set to be a negative value.

Q.65

When the rel_blk system call is executed, E_OBJ (–63) is returned each time, and the memory block is never returned. Why?

A.65

This phenomenon is unique to the RX850 Pro.

When the rel_blk system call is used to return a memory block, if the first four bytes of the memory block are not filled with zeros, the return value E_OBJ is used for termination instead of returning the memory block.

This specification was created in consideration of cases where a memory block is used as the message area for a mailbox.

When a memory block is used as the message area for a mailbox, the first four bytes become the message's wait queue link area.

In other words, when a message is queued in a mailbox and the rel_blk system call is issued, if the specification required that a memory block be returned, what would actually be returned would instead be the queue-linked message area.

To prevent this, the link area (first four bytes) must be all zeros in order for it to be regarded as a memory block that is being used as a message area, in which case the return value E_OBJ is used for termination instead of returning the memory block.

Therefore, when returning a memory block, the first four bytes must be cleared to zero.

However, starting in Ver. 3.15, in consideration of cases where the memory block is not used as a message area, a version of rel_blk that is able to return the memory block even when the first four bytes are not all zeros has been added as a separate library.

Function	Library Name
Library containing rel_blk that requires zero-clearing of first four bytes of memory block (same as previous specification for rel_blk)	librxp.a
Library containing rel_blk that does not require zero-clearing of first four bytes of memory block (new specification for rel_blk)	librxpm.a

These specifications of rel_blk are stored in separate libraries so that one or the other rel_blk specification can be used. Link to the library of the rel_blk specification to be used.

Q.66

In the GHS version, an endless loop occurs after the sample's memory initialization processing. Why?

A.66

In the GHS version's sample, there is a place where meminit.c is called from boot.850.

In meminit.c, RAM initialization processing is performed, but if processing cannot exit from that point or returning from initialization processing, operations are repeated from the start of boot.850.

This is caused by a damaged stack.

Since the stack area that is used in the startup routine is also zero-cleared in meminit.c, this type of phenomenon occurs.

Here are two fixes:

- Do not initialize stack areas that are used in startup routines.
- Temporarily save stack areas that are used in startup routines to a separate location.

For further description of stacks in startup routines, see Q.62.

Q.67

Where should the processing that is executed when an interrupt occurs be coded?

A.67

It should be coded in the interrupt handler.

The interrupt handler is a dedicated interrupt handling routine that is activated as soon as an interrupt occurs, and as such it is handled independently from tasks.

In the RX850 Pro, there are two types of interrupt handlers: a directly activated interrupt handler and an indirectly activated interrupt handler.

Q.68

What is the difference between the directly activated interrupt handler and the indirectly activated interrupt handler?

A.68

Directly activated interrupt handler:

When an interrupt has occurred, this interrupt handler is activated without going via the RX850 Pro.

However, the user is responsible for coding the register save processing and stack switching processing (a macro is provided for saving and restoring registers).

Indirectly activated interrupt handler:

When an interrupt has occurred, this interrupt handler is activated only after the RX850 Pro has performed interrupt preprocessing such as register save processing and stack switching processing.

Q.69

Which is better to use: the directly activated interrupt handler or the indirectly activated interrupt handler?

A.69

Since the directly activated interrupt handler starts handler processing as soon as an interrupt occurs, it can be expected to provide faster processing.

However, since users must themselves perform the processing to save contents of registers and stacks, this makes coding of the handler more complicated.

On the other hand, when an interrupt occurs, the indirectly activated interrupt handler shifts processing to the RX850 Pro, which saves the contents of registers and stacks, before passing processing to the handler.

Although the indirectly activated interrupt handler has somewhat slower responsiveness compared to the directly activated interrupt handler, it is simpler since it spares the user from having to code only the handler processing.

The user should take these factors into consideration when selecting which type of handler to use.

Q.70

How should the directly activated interrupt handler be registered?

A.70

The directly activated interrupt handler can be registered by assigning it to the handler address where control is passed by the processor when an interrupt occurs, or by setting a branch instruction to the directly activated interrupt handler.

For details of this programming, see **A.5 Directly Activated Interrupt Handler**.

Q.71

How should the indirectly activated interrupt handler be registered?

A.71

It is registered by being programmed in the CF definition file.

It can also be registered actively by using the def_int system call.

Q.72

What is the method for registering a timer interrupt?

A.72

Each timer is started as an indirectly activated interrupt handler.

For example, the source code appears as follows.

(CA850)

```
-------------------------
.section  "INTCMD"
RTOS_IntEntry_Indirect
-------------------------
```

(CCV850)

```
-------------------------
.org  0x00000240
RTOS_IntEntry_Indirect
-------------------------
```

RTOS_IntEntry_Indirect should be used as shown above.

This enables insertion of a timer interrupt.

Q.73

Are multiple interrupts supported?

A.73

Yes, they are.

Multiple interrupts are when one interrupt is inserted while another interrupt handler is being processed.

However, in the RX850 Pro, when an interrupt handler is started, its initial setting is “interrupt disabled,” so interrupts must be enabled within the interrupt handler in order to acknowledge multiple interrupts.

In other words, the user must exercise control as to whether interrupts will be enabled (set via EI or the ena_int system call) or disabled (set via DI or the dis_int system call).

Q.74

Can the #pragma rtos_interrupt in the NEC Electronics compiler (CA850) be used?

A.74

That is a pragma directive that is provided in order to simplify the coding of the real-time OS’s directly activated interrupt handler. Currently, however, normal operation is not possible when that pragma directive is used to program a directly activated interrupt handler.

Therefore, due to this restriction, it should not be used.

For details of programming the directly activated interrupt handler, see **A.5 Directly Activated Interrupt Handler**.

Q.75

What is the method for returning from an interrupt handler?

A.75

The return method differs for directly activated and indirectly activated interrupt handlers.

To return from a directly activated interrupt handler, use the macro that is provided in the handler's end section (the interrupt return processing is coded in this macro).

For description of the macro name and use method, see **A.5 Directly Activated Interrupt Handler**.

To return from an indirectly activated interrupt handler, use "return (TSK_NULL)" in the handler's end section if for ordinary return from an indirectly activated interrupt handler. Use "return (task ID number)" if a wake-up request is being issued for a task that is specified by a parameter.

Q.76

Although the following type of return value is required for indirectly activated interrupt handlers, what happens when registering a function in which "return (TSK_NULL)" has been deleted and there is no return value, as a handler?

```
ID
func_inthdr()
{
   /* Processing of indirectly activated interrupt handler */
   ..............................................
   ..............................................
   ..............................................

   /* Return processing from indirectly activated interrupt handler */
   return(TSK_NULL);
}
```

A.76

When registering a function that has no return value as a handler, there is a risk of a runaway condition.

Although return values are returned within the RX850 Pro when stored to r10, the r10 value is checked, and if it is not TSK_NULL (= 0), then the task that has the corresponding ID number is woken up.

In other words, operation is normal as long as 0 has been stored to r10, but when that is not the case an unexpected task may be woken up and linked to the ready queue and may actually be activated if it is a high-priority task.

Consequently, the desired operation may not occur unless return from an indirectly activated interrupt handler has been programmed.

Therefore, be sure to include either "return (TSK_NULL)" or "return (ID tskid)" at the end of the interrupt handler.

Q.77

Where is the interrupt handler's stack area reserved?

A.77

The interrupt handler's stack area is reserved at the location specified by the CF definition file.

The CF definition file specifies the stack size and reserved area (SPOL0 or SPOL1).

Code example:

```
-- System information
clktim  0x1
clkhdr  0x7
defstk  0x100
intstk  0x100:SPOL0 ← Stack information for interrupt handler
prttsk  0x1
prtsem  0x1
prtflg  0x1
prtmbx  0x1
prtmpl  0x1
```

However, the interrupt handler's stack is not used as soon as the interrupt has been inserted.

If the interrupt is inserted while a task is being performed, the register contents to be saved are pushed onto the stack of the interrupted task.

Afterward, the stack pointer (SP: r3) switches to the interrupt handler's stack (to be precise, the task's lp and sp values are pushed onto the interrupt stack).

In the case of multiple interrupts, since an interrupt is inserted during processing of another interrupt, the interrupt handler's stack is used.

In other words, the more multiple interrupts are inserted, the more the interrupt handler's stack gets consumed.

Q.78

When returning from a directly activated interrupt handler, the global pointer (GP) value becomes incorrect and a runaway condition occurs. Why?

A.78

When returning from a directly activated interrupt handler, it is possible that the GP value may get corrupted, which can then cause operation problems.

This is because directly activated interrupt handlers do not guarantee TP and GP values, so whenever these values must be used within the handler, they must be set at the start of the handler.

Code example for version that supports CA850:

```
------------------------------------------------------------
  #include

  ID
  inthdr_body( )
  {
      __asm("mov #__tp_TEXT, tp");
      __asm("mov #__gp_DATA, gp");

      /* Processing of directly activated interrupt handler func_inthdr */
      ....................................
      ....................................
      ....................................

      /* Return processing from directly activated interrupt handler func_inthdr */
      return tskid
  }
------------------------------------------------------------
```

For details, see **A.5 Directly Activated Interrupt Handler**.

Using macros (RTOS_IntEntry and RTOS_IntReturn) at the start and end of the directly activated interrupt handler can save and restore the GP value, but the value is not set.

This is because, due to the characteristics of the directly activated interrupt handler, it is not an interrupt handler that can be reliably controlled by the RX850 Pro.

When a timer handler is being activated, the GP (r4) value is used as temporary register.

If a direct interrupt is inserted at that time, the correct GP value will not be set and operation will be abnormal.

However, in the case of an indirectly activated interrupt handler, setting of a specific GP and specific TP are both done by the RX850 Pro.

For detailed description of this coding, see **A.6 Indirectly Activated Interrupt Handler**.

Q.79

If a system call is issued during non-maskable interrupt processing, what kind of operation occurs?

A.79

Since non-maskable interrupts are not subject to interrupt prioritization, they are acknowledged with higher priority than all other interrupts.

Consequently, NMIs are acknowledged even when operating in interrupt disabled mode.

In the RX850 Pro, management information and queues may be overwritten when system calls are issued.

If an NMI occurs while the RX850 Pro is processing management information and if a system call is issued during this processing, the management information may not be updated correctly and subsequent operations may lead to a runaway condition.

Therefore, in the RX850 Pro, if a system call is issued within a non-maskable interrupt handler, subsequent operations are not guaranteed.

Q.80

Interrupt handlers cannot be activated. Why?

A.80

The following causes are possible.

- **Has an interrupt actually been inserted?**

  Use the debugger to set a break point at “interrupt handler address (vector)” and check whether or not the target interrupt has been inserted.

  If it has not been inserted, it means there has been no physical insertion of an interrupt.

  In such cases, it may be necessary to check how hardware is being initialized.

  Check the interrupt control register settings (if the interrupt mask is open, etc.) and make sure an interrupt trigger has been set.

  After doing the above, if the interrupt still doesn’t occur, it may be due to a problem in the target device.

- **Does the interrupt source name (interrupt source number) in the CF definition file match?**

  When using an indirectly activated interrupt handler, the interrupt source name is specified in the CF definition file.

  Unlike in the RX850, this specification is not made as an interrupt source name per se but rather as a value calculated using the formula “target interrupt exception code – 0x80/0x10”.

  Make sure that this value is correct.

  Also, be sure to check the start address (start of function) for the interrupt handler that is activated when the interrupt is inserted.

  This does not have to be done when using a directly activated interrupt handler.

  When using a directly activated interrupt handler, directly code a jr instruction at the interrupt handler address so that processing will branch to the handler at that point.

  If none of the above solves the problem, it could be that the application did not download correctly or that the link was not established correctly.

Q.81

When a timer interrupt occurs, if a UART transmit or receive interrupt occurs for a directly activated interrupt handler, the program enters a runaway condition.

At the time when the UART transmit or receive interrupt occurs, the TP setting is expected to be TP = 0x00000400, but when checked it was found to be TP = 0xFFFFFFFF.

As a result, an illegal branch occurred to the interrupt vector area and a runaway condition ensued. Why did this happen?

A.81

The following cause is possible.

When using a directly activated interrupt handler in the RX850 Pro, the TP and GP values must be set at the start of the handler.

In other words, the TP and GP values to be used by the handler must be determined before starting the handler.

[Reference] Q.78

Q.82

During linkage, the message

“Warning : register r1 used as source register”

appears when using the RTOS_IntEntry macro in the directly activated interrupt handler. Why?

A.82

The RTOS_IntEntryIndirect macro that is provided with the RX850 Pro uses reserve register r1, and that causes this warning message to be output.

However, there is actually no problem and in this case the warning message can be ignored.

Output of this warning message can be prevented by specifying the -w option in as850 to suppress warning messages.

However, this option will suppress all warning messages, so it is better to overwrite part of macro.h as described below.

In macro.h, line 28:

```
----------------------------------------
/* go indirect interrupt handler */
ld.w   sbt_intent[r2], r1
jmp    [r1]
----------------------------------------
```

Overwrite as follows.

```
----------------------------------------
.option nowarning
ld.w    sbt_intent[r2], r1
jmp     [r1]
.option warning
----------------------------------------
```

Q.83

What is a cyclic handler?

A.83

A cyclic handler is a cyclical processing program available to users.

Among all cyclical processing programs, the cyclic handler has the least amount of overhead prior to start of execution.

Q.84

How is a cyclic handler created?

A.84

It can be created and registered by a CF definition file or a def_cyc system call.

Q.85

How many cyclic handlers can be created?

A.85

The number of cyclic handlers up to the specified maximum number can be created.

However, the highest value that can be specified as the maximum number is 32767, so up to 32767 cyclic handlers can be created.

Q.86

How are cyclic handlers started?

A.86

In a CF definition file, it is started when “activate” is specified as the initial status.

Also, it can be started or stopped by issuing an act_cyc system call during certain tasks.

Q.87

Does the cyclically activated handler get activated during interrupt enabled mode?
Does it get activated during interrupt disabled mode?

A.87

The RX850 Pro’s cyclically activated handler does get activated during interrupt enabled mode.

To set interrupt disabled mode within the cyclically activated handler, a DI instruction (to issue the dis_int system call) must be used at the start of the handler.

However, if an interrupt occurs after the cyclically activated handler is activated but before the DI instruction is executed, that interrupt will be inserted.

The inserted interrupt is subject to prioritization, however.

For details, see Q.111.

Q.88

What is the method for returning from a cyclically activated handler?

A.88

When using C language, return by issuing the return instruction in the last section of the cyclically activated handler.

When using assembly language, return by issuing a jmp [lp] instruction.

Q.89

Where are clock interrupt sources specified?

A.89

They are specified in the CF definition file.

However, specifying a clock interrupt in a CF definition file only registers the clock interrupt. To actually initialize the timer hardware, an initialization handler, initialization task, or startup routine must be programmed.

Code example:

```
-- System information
clktim  0x1          ← Basic clock cycle (unit: [ms])
clkhdr  0x7          ← Clock interrupt source number: (interrupt exception code - 0x80)/0x10
defstk  0x100
intstk  0x100:SPOL0
prttsk  0x1
prtsem  0x1
prtflg  0x1
prtmbx  0x1
prtmpl  0x1
```

Q.90

What is the shortest possible cyclic activation interval for the cyclically activated handler?

A.90

Although there is no specific restrictions, it is not possible to specify a value that is equal to or less than the “timer’s basic clock cycle (clktim)” as specified in the CF definition file.

In cases where the cyclically activated handler is activated at a short interval or a lot of processing is performed within the handler, a clock interrupt may be inserted before the cyclic handler’s main processing has been completed.

In such cases, there is a chance that the cyclically activated handler’s processing may get nested for continuation, which would keep other task processing from occurring and or otherwise not operate as expected.

Also, in cases where multiple cyclically activated handlers are activated at the same time, it should be noted with caution that their processing may not be completed before the next timer interrupt is inserted.

Q.91

The minimum unit for the cyclically activated handler's processing is 1 ms, so what can be done to set up processing that is completed within 100 μs?

A.91

The RX850 Pro's cyclically activated handler was designed to run on the basic clock, which is input at a frequency of about 1 ms.

For processing that is completed in less than 1 ms, use the directly activated interrupt handler instead.

However, if interrupt processing were to be performed at a cyclical rate of 100 μs, it would drastically impede the progress of the RX850 Pro's processing.

Q.92

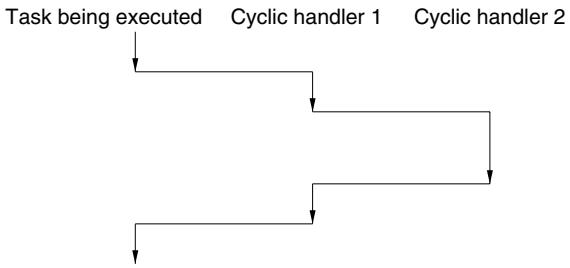
In a cyclically activated handler that is activated at an interval of 10 ms, what happens when processing that requires more than 10 ms is executed?

The interrupts are enabled in the cyclically activated handler.

A.92

In the RX850 Pro Ver. 3.13, when the timing requires the next cyclically activated handler be executed while the cyclically activated handler is still being executed, nesting is used to execute both cyclically activated handlers.

A transition diagram of this nesting is shown below.



Similar nesting for execution is used whether the cyclically activated handlers are different or the same.

Q.93

Is it correct that the interrupt processing is interrupted in a cyclically activated handler?

A.93

A cyclically activated handler can be activated during interrupt handler processing in the following cases.

- When the target interrupt handler is an interrupt that is activated with a lower priority than a timer interrupt
- When an EI section exists in the target interrupt handler
- When a timer interrupt is inserted before reaching the EI section, or when a timer interrupt is inserted during the EI section
- When the inserted timer interrupt is at the timing where a cyclically activated handler is activated

In cases such as these, the priority of the timer interrupt should be considered as dependent on the priority of the cyclically activated handler.

Q.94

When cyclically activated handlers are created and activated using the def_cyc system call, is the first activation of the cyclically activated handler timed to occur immediately after the def_cyc system call is issued? Or does this happen only after a specified amount of time has elapsed from when the def_cyc system call is issued?

A.94

It happens after a specified amount of time has elapsed from when the def_cyc system call is issued.

Similarly, the timing for cyclic handler activation by act_cyc also requires a specified amount of time to elapse after issuing the act_cyc.

Q.95

When multiple cyclically activated handlers are linked in a cyclically activated handler activation request queue, are tasks required to wait until all of the cyclically activated handler processing has been completed?

A.95

When multiple cyclically activated handlers are linked in a cyclically activated handler activation request queue, tasks are required to wait until all of the cyclically activated handler processing has been completed.

For details, see **7.6.6 Activation order of cyclically activated handler**.

Q.96

Is the following information correct with regard to the RX850 Pro interrupt management function's system calls?

	Maskable Interrupt	Scheduler Interrupt	Dispatch Processing
loc_cpu	Disabled	Disabled	Disabled
dis_int	Disabled	Disabled	Status before call is retained
unl_cpu	Enabled	Enabled	Enabled
ena_int	Enabled	Enabled	Status before call is retained

Maskable interrupt: Interrupt connected to the interrupt controller

Scheduler interrupt: Interrupt that triggers scheduling (periodic interrupt)

Dispatch processing: Scheduling that is triggered by a system call other than a scheduler interrupt

A.96

Yes, the above information is correct.

Q.97

When a timeout period has elapsed before a task is woken up due to fulfillment of a wake-up condition such as a twai_flg system call, is the return value E_OK? Or is it E_TMOUT?

Also, what happens if the fulfillment of a wake-up condition and the timeout happen in reverse order?

A.97

If the timeout occurs before the wake-up condition is met, the timeout results in a return value of E_TMOUT.

In other words, this reflects the sequence of events.

Accordingly, if the wake-up condition is met before the timeout occurs, the return value will be E_OK.

Q.98

What is the relation between clktim as defined in a CF definition file and the delay time of dly_tsk (dlytim)?

When 10 is set for clktim and dly_tsk (100) is set, the delay becomes one second, and the expected operation does not occur.

A.98

The value specified for clktim is actually expressed as the interval of the interrupt inserted by hardware, and the RX850 Pro uses this as the basis for calculating the time for the delay specified by dly_tsk.

In other words, if dly_tsk (1000) is specified, the task operation is extended by 1000 ms, but this does not indicate the timer interrupt interval in the RX850 Pro.

That is why clktim is used to specify the timer interrupt interval as a number of ms.

This enables the calculation of the number of interrupts needed for a 1000 ms delay.

Therefore, the expected delay time is not obtained when the actually input timer interval differs from the value specified by clktim.

Q.99

What is the RX850 Pro's timer precision?

A.99

The RX850 Pro's timer precision is at least 1 ms.

Although the timer precision itself depends on hardware settings, the minimum value that can be specified as the "basic clock cycle (clktim)" that is specified in the CF definition file is 1 ms.

This value must be combined with the timer cycle that is actually set by hardware.

When the RX850 Pro performs time-related processing, this processing is based on the numbers (in ms units) that are specified by system call arguments.

For example, when dly_tsk (1000) is specified, it sets a 1000 ms delay for task processing, and this 1000 ms value was calculated based on the value specified by clktim.

In other words, if "1" is specified as "clktim", the RX850 Pro interprets that as meaning that 1000 timer interrupts need to be inserted to create a 1000 ms delay.

If "5" is specified as "clktim", 200 timer interrupts need to be inserted to create a 1000 ms delay.

Note with caution when migrating from the RX850 to the RX850 Pro that this calculation method differs from that used in the RX850.

In the RX850, the value specified as the argument is the "number of the timer interrupts inserted (tick)".

Q.100

This question concerns the clock handler cycle.

In the AZ850's Analyze Window, the cycle of the output INTCM40 was approximately 0.113 ms.

The reset.c timer was set as follows.

TMC40 = 0x86;

CM40 = 10;

CMIC40 = 0x0;

In the V850E/MS1, the expected settings are ϕm/16 for the internal count clock value, ϕ/8 for the intermediate clock value, and 10 for the count value, but the output result did not match the calculated result.

How should that be interpreted?

A.100

It means there is an error in a register setting.

In the V850E/MS1, when TMC40 = 0x86 is set via a register, the internal count clock value becomes ϕ/32. The internal system clock is calculated as 25 MHz, so that the cycle of INTCM40 (timer 4 interval) becomes approximately 0.113 ms.

The specific calculation steps are described below.

1. The timer control register (TMC40) settings are determined based on the selected internal count clock and intermediate clock.
   When TMC40 = 0x86 (10000110), the internal count clock setting is ϕm/32 and the intermediate clock setting has been selected as ϕ/8.

[Overview of timer control register (TMC40)]

	7	6	5	4	3	2	1	0
TMC40	CE40	0	0	0	0	PRS400	PRM401	PRM400

Bit position	Bit name	Meaning
7	CE40	Controls timer operation 0: No operation 1: Count operation
2	PRS400	Internal count clock (ϕm: intermediate clock) 0: ϕm/16 1: ϕm/32
1, 0	PRM401, PRM400	Intermediate clock ϕm (ϕ: internal system clock)

PRM401	PRM400	ϕm
0	0	ϕ/2
0	1	ϕ/4
1	0	ϕ/8
1	1	RFU (reserved)

2. The interval time is determined using the following formula.

(Interval time) = (Compare register value + 1) × (Count clock cycle)

The count clock cycle is determined using the following formula.

(Count clock cycle) = 1/(Internal count clock)

When the internal system clock ϕ = 25 [MHz] = 25×10^6 [Hz], the count clock cycle is as shown below.

$$1/(\phi/8 \times 1/32) = 1/\{25 \times 10^6 \times 1/8 \times 1/32\}$$
$$= (8 \times 32)/(25 \times 10^6)$$

Consequently, when the compare register CM40 = 10, the interval time is determined as follows.

$(10 + 1) \times \{(8 \times 32)/(25 \times 10^6)$ [Hz]$\} \fallingdotseq 0.113$ [ms]

[Reference] Q.101

Q.101

What settings should be made in order to enable use of timer 4 as an interval timer?

A.101

The following description uses the V850E/MS1 as an example.

When the internal system clock is 25 MHz, the timer control register (TMC40) and compare register (CM40) values are set so that the interval time becomes approximately 0.1 ms.

1. Select internal count clock and intermediate clock and set the timer control register (TMC40) value.

Example:

Internal count clock → ϕm/32
Intermediate clock → ϕ/8

	7	6	5	4	3	2	1	0
TMC40	1	0	0	0	0	1	1	0

[Overview of timer control register (TMC40)]

	7	6	5	4	3	2	1	0
TMC40	CE40	0	0	0	0	PRS400	PRM401	PRM400

Bit position	Bit name	Meaning
7	CE40	Controls timer operation 0: No operation 1: Count operation
2	PRS400	Internal count clock (ϕm: intermediate clock) 0: ϕm/16 1: ϕm/32
1, 0	PRM401, PRM400	Intermediate clock ϕm (ϕ: internal system clock)

PRM401	PRM400	ϕm
0	0	ϕ/2
0	1	ϕ/4
1	0	ϕ/8
1	1	RFU (reserved)

2. Set the compare register (CM40) value.
The interval time is determined using the following formula.

(Interval time) = (Compare register value + 1) × (Count clock cycle)

1) Determine the count clock cycle.
The count clock cycle is determined using the following formula.

(Count clock cycle) = 1/(Internal count clock)

When the internal system clock ϕ = 25 [MHz] = 25×10^6 [Hz], the count clock cycle is as shown below.

$$1/(\phi/8 \times 1/32) = 1/\{25 \times 10^6 \times 1/8 \times 1/32\}$$
$$= (8 \times 32)/(25 \times 10^6)$$

2) Determine the compare register value (compare register value: n).
When the interval time is 0.1 [ms] = 1×10^{-4} [s], the compare register value is calculated as follows.

$$1 \times 10^{-4} \text{ [s]} = (n + 1) \times \{(8 \times 32)/(25 \times 10^6) \text{ [Hz]}\}$$
$$n = 1 \times 10^{-4} \text{ [s]} \times \{(25 \times 10^6) \text{ [Hz]}/(8 \times 32)\} - 1 \fallingdotseq 9$$

Consequently, to set the interval time as a value of approximately 0.1 ms, 9 should be set as the CM40 register value.
Since the interval time is 0.1024 ms when CM40 = 9, the value to be expected from the calculation is approximately 0.1 ms.

Q.102

The error message “fa01 (F): PC location’s line information not found” appears. Why?

A.102

When the Source Window appears after a program has been stopped, it displays the source file that corresponds to the program counter (PC) value at the time the program was stopped.

This error message indicates that the source file corresponding to that program counter (PC) value cannot be found.

The following causes are possible.

(1) Source file exists somewhere outside of the source path.
(2) The program was stopped at a location where there is no corresponding source file, such as at a library or real-time OS system call.
(3) The program had entered a runaway condition, and the execution had stopped after branching to an address not used by the program.
(4) No build was done in debug mode, so the debug information is not included in the object.

If the cause is (1) above, the source path is the directory where the load module files being downloaded are stored by default.

If the source file is located in any other directory, go to the [Option] menu and select [Debugger options] to specify the source path.

If the cause is (2) above, and if display of this message cannot be avoided due to the structure of the program, use the Console Window to turn off display of error dialog boxes.

[Reference] Q.114

Q.103

During linkage, the message
“multiple inclusion of same file attempted, ignored.”
appears. Why?

A.103

This message is displayed when the source for an object that is registered as a startup file has been registered as the source to be assembled.

In other words, start.o has been registered as the startup file and its source file, start.s, is to be registered as the object to be assembled (for the RX850 Pro, these files are boot.o instead of start.o and boot.s instead of start.s).

In the dialog box used to set up projects for PM, delete start.s from the “Source file name” list and then use the Linker Options Setup dialog box to change start.s to start.o as the “Startup file” setting.

Also, before creating objects for ROM programming, the same phenomenon occurs when creating code that is used to reserve ROM programming space.

As with start.o files, the method for fixing this is to build without registering the object as a source file.

This type of operation is especially needed for applications that use a real-time OS, when it is essential to use a particular startup module.

Q.104

During linkage, the error message
“undefined: ’__e_sysfnc’ referenced in ’c:\nectools32\lib850e_ghs\r32\rxcore.o’”
appears. Why?

A.104

This message was displayed because svc.o, which is used to assemble svc.s, was not linked.

This file contains a system call table that was created by the configurator cf850pro.

This table’s start address is __e_sysfunc.

Note with caution that this file is easily overlooked, since it is not required by the RX850 V3.1x versions.

System calls to be used must be specified in a CF definition file.

For description of the specification method and configurator startup method, see **6.6 Specification Format for SCT Information** and **CHAPTERS 7** and **8 OPERATING CONFIGURATOR** in the **RX850 Pro Installation User’s Manual**.

Q.105

The message “Warning:address is too long” appears after using PM plus to build when hx850 is executed. Why?

A.105

This is due to a limitation of the Intel hexadecimal format. This message is always displayed when the Intel hexadecimal format has been specified and the program to be run extends over 1 MB of space.

During the actual development, this should be considered when converting load modules to hex format, but otherwise this message can be ignored.

This message will not appear if Motorola hexadecimal format is selected instead of Intel hexadecimal format.

Q.106

Why has the system entered a runaway condition?

A.106

The following causes are possible.

- Are the task stack area and interrupt stack area (system stack area) large enough?
  Insufficient stack areas account for most causes of poor operation among applications that use a real-time OS.
  Specifically, when a task stack or interrupt stack exceeds its specified size, other task's stacks can be damaged, or the RX850 Pro's system management area can be damaged, and in either case a runaway condition may result.
  Note also that, with the RX850 Pro, incorrect values included in addresses are not questioned in order to ensure reliable operation of the address information.

- Is the indirectly activated interrupt handler's termination processing coded correctly?
  If the indirectly activated interrupt handler's termination processing, which means the setup for the return value, is not correct, processing after an interrupt may become unreliable.
  This is not always the case: sometimes processing happens to continue correctly.
  When returning from the indirectly activated interrupt handler, be sure to specify "TSK_NULL" as an argument, such as "return (TSK_NULL);".
  Or, if another task should be woken up after returning from the indirectly activated interrupt handler, specify the task ID of the target task as an argument, such as "return (TASK_ID);".

- If the task does not have an endless loop specification (for (; ;), while (1)), is an ext_tsk system call being issued to perform the termination processing?
  This is not a problem if the task has an endless loop specification, but if it does not then the task must be terminated by issuing the ext_tsk system call at the end.
  When a system call is not specified for a task with no endless loop specification, then the RX850 Pro is unable to determine whether or not the task has been terminated.
  This will definitely result in a runaway condition.

Q.107

The boot processing itself works well, and control is passed to the RX850 Pro after boot processing is completed, but afterward no tasks can be performed, and the system gets stuck in HALT mode. Why?

A.107

This can happen when problems occur in the RX850 Pro's initialization program.

When control is passed to the RX850 Pro after boot processing, other initialization tasks are started, such as the creation of a system base table (SBT).

At that point, various management tables and memory pools are created, but the system may be set to HALT mode if a failure occurs when attempting to make them.

Causes for such failures include not having a reserved system memory pool (SPOL0) available for creating the SBT and management tables, which can occur when memory space is either write-protected or insufficient.

Q.108

Why doesn't the expected operation occur after issuing a system call?

A.108

The following causes are possible.

- Causes related to task priority
  When a wait mode is cleared and a system call is issued, if the priority of the task released from wait mode is lower than that of other tasks, the released task may not be performed immediately as expected.

- Causes related to stack damage
  If wait-related system calls such as wai_flg and wai_sem are issued but problems occur when their wait modes are cleared, this can result in damage to task stack or interrupt stack areas.
  If either task stack or interrupt stack areas grow beyond their set values, they can damage the RX850 Pro management area and the wait information it contains.

Q.109

The return value for a system call was "–17".
What does this error value mean?

A.109

In this case, E_NOSPT is indicated as the type of error.
It means that the system call that was issued has not been registered in the system call table.

Q.110

I understand about linking the RX850 Pro's nucleus object rxcore.o (rxtmcore.o), but I have not been able to register the object to the file list using PM plus.
How can I link this file?

A.110

When specifying an object to be linked using PM plus, go to the "Other" part of the Linker Options Setup dialog box and enter the object name where it says "Other option (Y)".

For safety's sake, also enter the file path.

For example, to link rxcore.o, specify the following (when using rxtmcore.o, specify "c:nectools32\lib850e\r32\rxtmcore.o").

```
c:\nectools32\lib850e\r32\rxcore.o
```

Use a space to separate multiple objects.

Do not use a ": (colon)" or "; (semicolon)" as a separator.

Q.111

What is the difference between nucleus objects rxcore.o and rxtmcore.o?

A.111

They differ in the way they acknowledge interrupts during processing of timer interrupts.

Object Name	Description
rxcore.o	Nucleus kernel that can acknowledge interrupts at all interrupt levels during cyclically activated handler processing
rxtmcore.o	Nucleus kernel that can acknowledge only interrupts with a higher priority than timer interrupts during cyclically activated handler processing

The cyclically activated handler is called from the timer handler.

With rxcore.o, interrupt termination processing (reti) occurs once during execution of the timer handler.

Therefore, the interrupts at all interrupt levels can be acknowledged during processing of the cyclically activated handler.

However, with rxtmcore.o, the cyclically activated handler is called during execution of the timer handler, so only interrupts with a higher priority than timer interrupts can be acknowledged.

Q.112

In the RX850 Ver. 3.1, the .sit section had to be placed within ±32 KB of address 0. Is this also the case in the RX850 Pro?

A.112

This restriction does not exist in the RX850 Pro.

There are no restrictions on placement locations.

Q.113

How can I program ROM or FLASH ROM using only the RX850 Pro?

A.113

One method is to first program the RX850 Pro software, then make necessary revisions in user applications only.

This is enabled by separating ROM-programmed sections.

The sections available for ROM programming in the RX850 Pro are listed below.

- .system
- .system_int
- .system_cmn

Software that can be placed in the .system section includes the following.

- Routines commonly used by RX in rxcore.o
- svc.o (system call table)
- System call main programs (cretsk.o, etc.)
- Routines commonly used with system calls (f_memget.o, etc.)

Software that can be placed in the .system_int section includes the following.

- Interrupt processing programs within rxcore.o

Software that can be placed in the .system_cmn section includes the following.

- Scheduler processing programs within rxcore.o

However, the system call main programs that may be used in the future even if they are currently not used must also be placed.

For information on which objects are actually used from among those in the libraries (librxp.a and librxpm.a), use the linker options to specify the link map output (NEC Electronics version: -m, GHS version: -map) and acquire the link information, then refer to the object names that are output.

Once the above countermeasures are performed, the ROM-programmed area in the RX850 Pro is no longer affected by changes in user programs.

The interface library and .sit section should be linked to the user application side.

The user application side's boot sections (boot.s and boot.850) includes the following code.

```
mov #__rx_start, lp
jmp [lp]
```

The __rx_start symbol is on the ROM-programmed RX850 Pro's side, so a jmp instruction is required after setting the actual address as #__rx_start.

The following shows examples of link directives for these sections (NEC Electronics version).

[Example 1] Method for setting separate output sections for objects in libraries

```
SYSTEM : !LOAD ?RX {
  .system_svc = $PROGBITS ?AX .system { svc.o };
  .system_core = $PROGBITS ?AX .system { ..\..\..\lib850e\r32\rxcore.o };
  .os_lib1 = $PROGBITS ?AX .system {udfsys.o(c:\nectools32\lib850e\r32\librxp.a)};
  .os_lib2 = $PROGBITS ?AX .system {relblk.o(c:\nectools32\lib850e\r32\librxp.a)};
  .os_lib3 = $PROGBITS ?AX .system {getblk.o(c:\nectools32\lib850e\r32\librxp.a)};
  .os_lib4 = $PROGBITS ?AX .system {gettim.o(c:\nectools32\lib850e\r32\librxp.a)};
  .os_lib5 = $PROGBITS ?AX .system {sndmsg.o(c:\nectools32\lib850e\r32\librxp.a)};
  .os_lib6 = $PROGBITS ?AX .system {rcvmsg.o(c:\nectools32\lib850e\r32\librxp.a)};
  .os_lib7 = $PROGBITS ?AX .system {sigsem.o(c:\nectools32\lib850e\r32\librxp.a)};
  .os_lib8 = $PROGBITS ?AX .system {waisem.o(c:\nectools32\lib850e\r32\librxp.a)};
  .os_lib9 = $PROGBITS ?AX .system {setflg.o(c:\nectools32\lib850e\r32\librxp.a)};
  .os_lib10 = $PROGBITS ?AX .system {waiflg.o(c:\nectools32\lib850e\r32\librxp.a)};
  .os_lib11 = $PROGBITS ?AX .system {wuptsk.o(c:\nectools32\lib850e\r32\librxp.a)};
  .os_lib12 = $PROGBITS ?AX .system {statsk.o(c:\nectools32\lib850e\r32\librxp.a)};
  .os_lib13 = $PROGBITS ?AX .system {exdtsk.o(c:\nectools32\lib850e\r32\librxp.a)};
  .os_lib14 = $PROGBITS ?AX .system {exttsk.o(c:\nectools32\lib850e\r32\librxp.a)};
  .os_lib15 = $PROGBITS ?AX .system {cretsk.o(c:\nectools32\lib850e\r32\librxp.a)};
  .system_cmn = $PROGBITS ?AX .system_cmn;
  .system_int = $PROGBITS ?AX .system_int;
};
TEXT : !LOAD ?RX {
 .text = $PROGBITS ?AX .text;
};
```

[Example 2] Method for setting one section for all objects in libraries

```
SYSTEM : !LOAD ?RX {
  .system_svc = $PROGBITS ?AX .system { svc.o };
  .system_core = $PROGBITS ?AX .system { ..\..\..\lib850e\r32\rxcore.o };
  .system = $PROGBITS ?AX .system { svc.o ..\..\..\lib850e\r32\rxcore.o
    udfsys.o(c:\nectools32\lib850e\r32\librxp.a)
    relblk.o(c:\nectools32\lib850e\r32\librxp.a)
    getblk.o(c:\nectools32\lib850e\r32\librxp.a)
    gettim.o(c:\nectools32\lib850e\r32\librxp.a)
    sndmsg.o(c:\nectools32\lib850e\r32\librxp.a)
    rcvmsg.o(c:\nectools32\lib850e\r32\librxp.a)
    sigsem.o(c:\nectools32\lib850e\r32\librxp.a)
    waisem.o(c:\nectools32\lib850e\r32\librxp.a)
    setflg.o(c:\nectools32\lib850e\r32\librxp.a)
    waiflg.o(c:\nectools32\lib850e\r32\librxp.a)
    wuptsk.o(c:\nectools32\lib850e\r32\librxp.a)
    statsk.o(c:\nectools32\lib850e\r32\librxp.a)
    exdtsk.o(c:\nectools32\lib850e\r32\librxp.a)
    exttsk.o(c:\nectools32\lib850e\r32\librxp.a)
    cretsk.o(c:\nectools32\lib850e\r32\librxp.a)
  };
  .system_cmn = $PROGBITS ?AX .system_cmn;
  .system_int = $PROGBITS ?AX .system_int;
};
TEXT : !LOAD ?RX {
  .text = $PROGBITS ?AX .text;
};
```

Both of the above examples are just general examples.

During linkage, be sure to add an option to reference librxp.a (or librxpm.a).

Q.114

The external RAM that is implemented in our target system is accessible in both 16-bit and 32-bit units but cannot be accessed in 8-bit (1-byte) units.

Can the RX850 Pro be used with this system?

A.114

The SPOL0 area assigned to the RX850 Pro's management area requires 8-bit (1-byte) accessibility for data access. In other words, in cases where 8-bit access is not enabled in the external RAM area, the SPOL0 area cannot be placed in that area. If it is placed there, data being operated on would be lost and normal operations could not be guaranteed.

However, since there is no 8-bit access to stacks or memory pools, there would be no problem using the SPOL1 area.

Nevertheless, any code that accesses data in memory blocks created within the SPOL1 area would have to have 8-bit access suppressed, such as via a compile option.

This type of problem can be resolved by placing the SPOL0 area within the internal RAM of any V850 Series device.

APPENDIX C INDEX

[T]

[U]

[V]

[W]

APPENDIX D REVISION HISTORY

A history of the revisions up to this edition is shown below. “Applied to” indicates the chapters to which the revision was applied.

Edition	Contents	Applied to:
2nd	Modification of return value E_NOSPT from –11 to –17	Throughout
	Modification of description on target CPU	CHAPTER 1 OVERVIEW
	Modification of description on hardware and software environments	
	Addition of explanation on system construction procedure	
	Addition of a caution on return processing from directly activated interrupt handler	CHAPTER 5 INTERRUPT MANAGEMENT FUNCTION
	Addition of a caution on changing/acquiring interrupt control register	
	Modification of description on overview of memory management function	CHAPTER 6 MEMORY POOL MANAGEMENT FUNCTION
	Modification of description on management objects	
	Modification of description on memory pool and memory blocks	
	Addition of a caution on returning a memory block	
	Addition of explanation on interrupt during cyclically activated handler	CHAPTER 7 TIME MANAGEMENT FUNCTION
	Addition of explanation on activation order of cyclically activated handler	
	Addition of explanation on idle handler	CHAPTER 8 SCHEDULER
	Addition of explanation on system initialization processing	CHAPTER 9 SYSTEM INITIALIZATION
	Addition of explanation on directly activated interrupt handler	APPENDIX A PROGRAMMING METHODS
	Addition of explanation on indirectly activated interrupt handler	
★ 3rd	Modification of V850 Family to V850 Series	Throughout
	Addition of operating target CPU	CHAPTER 1 OVERVIEW
	Modification of description on peripheral controller	
	Addition of target device for in-circuit emulator	
	Addition of I/O board for in-circuit emulator and target device	
	Modification of description on software environment OS and addition of debugger	
	Modification of description of caution on returning a memory block	CHAPTER 6 MEMORY POOL MANAGEMENT FUNCTION
	Modification of description on interrupts in cyclically activated handler	CHAPTER 7 TIME MANAGEMENT FUNCTION
	Modification of caution in description of rel_blk to match caution on returning a memory block	CHAPTER 11 SYSTEM CALLS
	Modification of systime to t_systime in description of structure of system clock SYSTIME for set_tim and get_tim	
	Addition of APPENDIX B Q & A	APPENDIX B Q & A