USER'S MANUAL

NEC

78K/IV SERIES REAL-TIME OS

RX78K/IV

FUNDAMENTAL

Document No. U10603EJ3V0UM00 (3rd edition)
Date Published January 1997 N
Printed in Japan

TRON is an abbreviation of The Realtime Operating system Nucleus.

ITRON is an abbreviation of Industrial TRON.

MS-DOS is a trademark of Microsoft Corporation.

IBM-PC/AT is a trademark of IBM Corporation.

The other company names and product names appearing in this document are the trademarks of the respective companies.

M7A 96.10

Regional Information

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

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

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

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

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

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

NEC Electronics Italiana s.r.1.
Milano, Italy
Tel: 02-66 75 41
Fax: 02-66 75 42 99

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

NEC Electronics (France) S.A.
Velizy-Villacoublay, France
Tel: 01-30-67 58 00
Fax: 01-30-67 58 99

NEC Electronics (France) S.A.
Spain Office
Madrid, Spain
Tel: 01-504-2787
Fax: 01-504-2860

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

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

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

NEC Electronics Singapore Pte. Ltd.
United Square, Singapore 1130
Tel: 253-8311
Fax: 250-3583

NEC Electronics Taiwan Ltd.
Taipei, Taiwan
Tel: 02-719-2377
Fax: 02-719-5951

NEC do Brasil S.A.
Sao Paulo-SP, Brasil
Tel: 011-889-1680
Fax: 011-889-1689

INTRODUCTION

Thank you for buying NEC's 78K/IV Series Real-time OS RX78K/IV, which is embedded software for the 78K/IV Series.

This manual is intended to introduce to the user the general idea of 78K/IV series real-time OS RX78K/IV.

[Readers]

It is assumed that the readers of this manual have already read the user's manual of the device and has experience in software programming.

Note that this package only contains the real-time OS. To use the real-time OS actually, the "CC78K4 series C-compiler" and "RA78K4 series assembler package" are necessary.

[Organization]

This manual is organized as follows:

CHAPTER 1 GENERAL

Describes general descriptions and execution environment of this real-time OS.

CHAPTER 2 BASIC FUNCTIONS OF RX78K/IV

Describes the basic functions of the real-time OS.

CHAPTER 3 RX78K/IV BASIC OPERATIONS AND CONTROL BLOCKS

Describes the operation and control block of the real-time OS.

CHAPTER 4 RX78K/IV SYSTEM CALL LIST

Describes system calls supplied by the real-time OS.

CHAPTER 5 RX78K/IV RESET OPERATION

Describes the operation to initialize this real-time OS.

APPENDIX 1 ERROR CODE LIST AND TASK STATUS LIST

Describes the return parameters of the system calls and task status.

APPENDIX 2 DESCRIPTION IN C LANGUAGE

Describes the functions supported by C compiler "CC78K4".

APPENDIX 3 SYSTEM CALL LIST

Lists the system calls supported by this real-time OS.

APPENDIX 4 EXAMPLE OF CYCLIC HANDLER CODING

Shows examples of describing cyclic handlers.

APPENDIXES 5 AND 6 ESTIMATING MEMORY CAPACITY

Describes how to estimate the memory capacity used by this real-time OS, taking examples.

APPENDIX 7 STANDBY FUNCTION

Describes the method to implement the standby function supplied by the 78K/IV by using this real-time OS.

APPENDIX 8 MAXIMUM STACK SIZE USED BY EACH SYSTEM CALL

Indicates the stack capacity used when issuing each system call.

APPENDIX 9 RESTRICTIONS OF SYMBOL NAMES

Explains the symbol names used by this real-time OS.

[Legend]

The following symbols are used throughout this manual:

… : Repeat
[ ] : Can be omitted
“ ” : Character string
‘ ’ : Character string
( ) : Character
Bold : Character as is
_ : Important point. Underlined characters in application example must be input as is.
∆ : One or more blank
: : Omission of program description
/ : Delimiter
\ : Back slash

[Related documents]

The following table lists the documents (user’s manuals) related to this manual.

Document Name	Document No.
78K/IV series Real-time OS RX78K/IV -Install	U10604EJ3V0UM00
78K/IV series Real-time OS RX78K/IV -Debugger	U10364EJ1V0UM00
RA78K4 Assembler package -Language	U10334EJ1V0UM00
RA78K4 Assembler package -Operation	U11162EJ1V0UM00
CC78K4 C compiler -Language	—
CC78K4 C compiler -Operation	—
78K/IV series -Instruction	EEU-1386

CONTENTS

LIST OF FIGURES

LIST OF TABLES

[MEMO]

CHAPTER 1 GENERAL

Systems in the field of control equipment are required to respond immediately to changes in external and internal events. The conventional systems have tried to satisfy this requirement by using simple interrupt programs. As the application systems have increasingly become high-performance and sophisticated, however, many problems that cannot be solved only by simple interrupt programs have arisen.

In other words, as systems have increasingly become complicated and as the quantity of the programs to be executed has increased, it has been difficult to manage the execution sequence ofthe programs.

The real-time operating system (hereafter referred to as the “real-time OS”) was developed to cope with this problem. The real-time OS immediately respond to changes in an event and executes the optimum programs in the optimum sequence.

1.1 Multi-task OS

The minimum unit that is executed under the management of the real-time OS is called a task. Multi-tasking is executing two or more tasks simultaneously on one CPU.

Actually, however, the CPU can only execute one program at a time. If two or more tasks are selected and executed alternately, for example, at fixed time intervals, however, it seems to the human eye as if these tasks are being executed at the same time.

The tasks are executed by taking advantage of opportunities such as time intervals. An OS that has a function to execute tasks in parallel by selecting a task at specific opportunities is called a multi-task OS. A multi-task OS is intended to improve the overall processing ability of the system processing tasks in parallel.

1.2 RX78K/IV

The RX78K/IV is a real-time OS for embedded control applications and has been developed to supply an efficient real-time/multi-task processing environment and to expand the application range of a targeted CPU in the control equipment field, providing complete real-time/multi-task functions.

Because it is assumed that the RX78K/IV is embedded in the target system, it has been designed as a compact, high execution speed OS, for implementation in ROM.

1.3 Outline and Features of RX78K/IV

(1) Conforms to μITRON Ver.2.01

The system calls of the RX78K/IV conform to the μITRON specifications which are the specifications for a real-time OS for embedded control applications.

(2) Real-time/multi-task processing functions

Many functions are provided to realize complete real-time/multi-task processing.

- Event-driven priority scheduling
- Three types of synchronization/communication functions
- Memory management function
- Interrupt management function
- Time management function

(3) Designed for ROMization

Because the RX78K/IV is an operating system intended for embedded control application, it is designed to be compact. System calls not used can be deleted when configuring a system.

(4) Convenient utilities for system configuration

Two utilities useful for configuring system are provided.

- Configurator
- High-level language interface

(5) RX78K/IV supports the following cross tools for 78K/IV series.

- RA78K4
- CC78K4

(6) Two types of memory models supported

The RX78K/IV supports the memory models supported by the CC78K4 as follows:

- Large model (1M-byte space) : supports location 0fh
- Small model (64K-byte space) : supports location 00h

1.4 Configuration and Size of RX78K/IV

The RX78K/IV consists of the following three subsystems.

(1) Nucleus

This is the central part of the RX78K/IV. It is embedded in the target system along with the application program and actually carries out real-time/multi-task control. It also performs processing in response to system calls issued from processing tasks.

The size of the nucleus differs depending on the system calls to be linked. In the case of the large model, it is about 7K bytes MAX.; in the case of the small model, it is about 5K bytes MAX.

(2) C language interface library

When the user program is described in C language, system calls are called in the format of C language calling functions. This format, however, differs from the input format the nucleus can understand.

The C language interface library is a program that converts system calls written in C into an input format the nucleus can understand and links the processing tasks described in C with the nucleus.

The C language interface library is described so that it can be used with the NEC's 78K/IV series C compiler CC78K4.

(3) Configurator

This is a tool for creating tables with information on the user system necessary for the nucleus.

The configurator runs on the development machine and can be used to input and correct the information on the user system necessary for the OS in interactive mode.

1.5 Memory Confuguration of RX78K/IV

The memory of the system using RX78K/IV consists of the following:

(1) Nucleus
(2) Reset routine for system initialization
(3) System call entry address storage table
(4) OS management area[Note]
(5) Initialization information table
(6) Object management control block[Note]
(7) User program

[Large model]

(1), (2), and (3) above are stored in the internal ROM or external memory space at addresses 0H through 0FFFFH. Because (1), (2), and (3) are relocatable objects, they can located in any area of memory in the above range.

(4), (5), and (6) are information tables that are referenced by the reset routine when it initializes the system (these tables are created by the configurator).

The user can locate these tables in any area.

Note: The area in which (4) and (6) can be located is within 64K bytes with the most significant byte fixed. The area is allocated starting from the address specified by the user.

[Small model]

(1), (2), and (3) above are stored in the internal ROM or external memory space at addresses 0H through 0FFFFH. Because (1), (2), and (3) are relocatable objects, they can located in any area of memory in the above range.

(4), (5), and (6) are information tables that are referenced by the reset routine when it initializes the system (these tables are created by the configurator).

Locate (4) and (6) within an 256-byte area at addresses 0F700H through 0F7FFH in the internal RAM area.

For the memory map, refer to the User's Manual of the CPU you are using.

An example of memory configuration when the RX78K/IV is embedded with the μPD784026 is shown below.

Fig. 1-1 Memory Structure Example (Large Model)

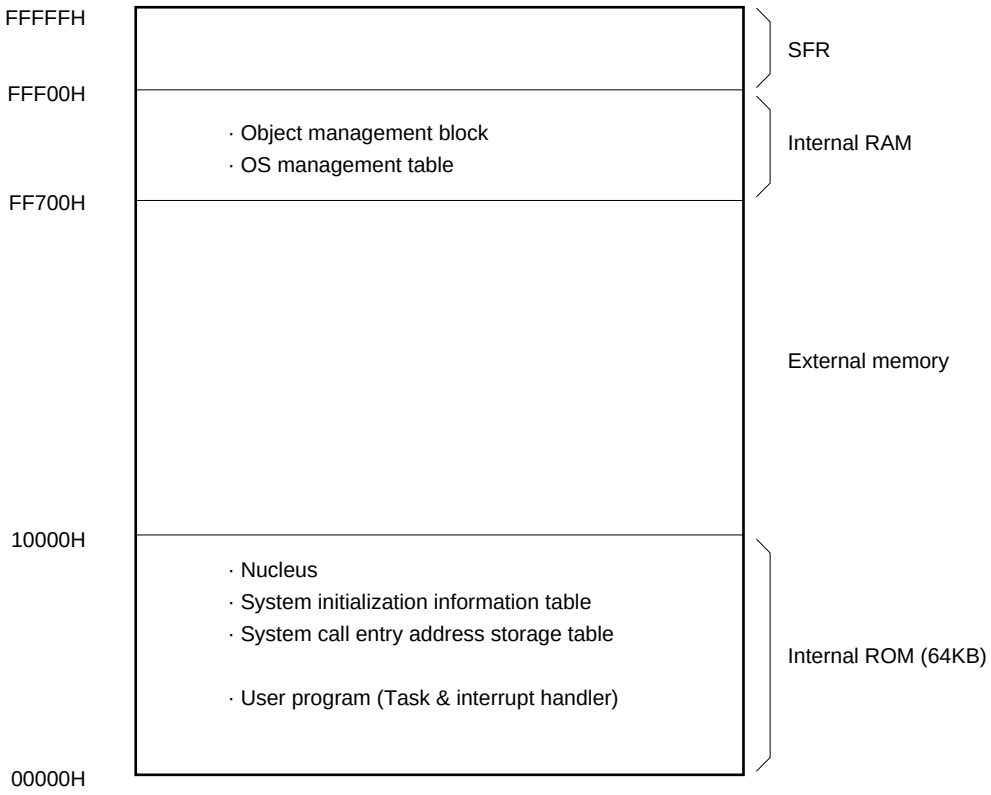
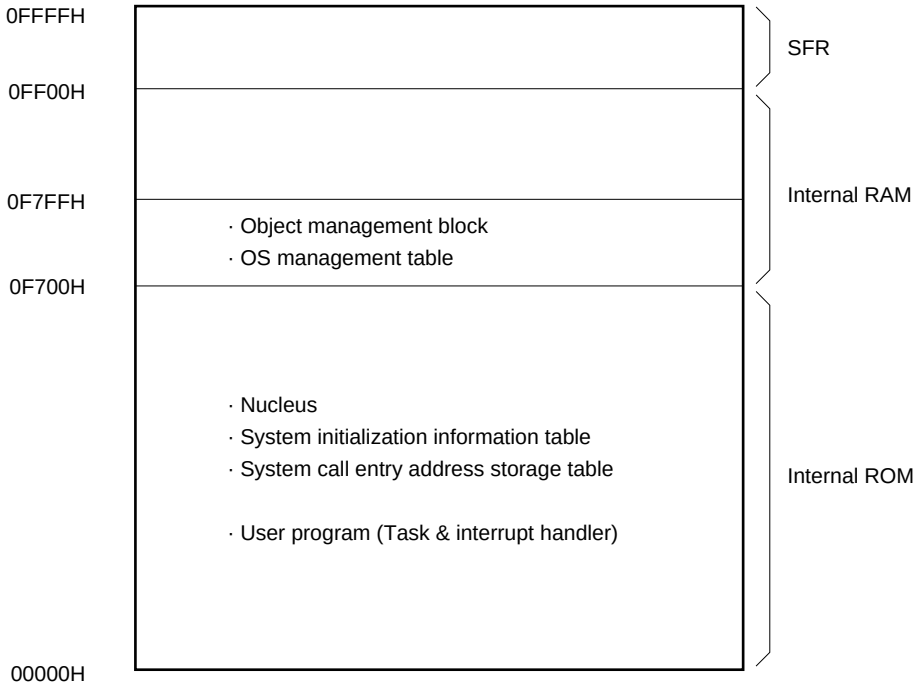


Fig.1-2 Memory Structure Example (Small Model)



[MEMO]

CHAPTER 2 BASIC FUNCTIONS OF RX78K/IV

Because RX78K/IV is a multi-task OS, it can process two or more tasks concurrently.

78K/IV series microcomputer has 8 register banks. With RX78K/IV, bank 0 of these register banks is used by the OS. Tanks can be allocated to registers for high-speed task switching. (For details, refer to **3.1 RX78K/IV Configuration)**

There are 16 levels of task priorities (0 to 15), with the lowest number assigned the highest priority level. However, priority 0 and priority 1 are reserved for OS. Therefore, priorities to which user tasks can be allocated are limited to the 14 levels of 2 to 15.

This chapter describes the following items of the basic functions:

(1) Task state
(2) Task manipulation
(3) Synchronization and communication
(4) Interrupt processing
(5) Memory management
(6) Time management

2.1 Task State

A task can be in any of the following 4 states:

(1) RUN state

In this state, the task is given the privilege to use the CPU and is under execution. Only one task in the RUN state can exist in the system at a given point of time.

(2) READY state

A task in the state is ready for execution. All the elements to place the task in the RUN state are ready, but the task in the READY state cannot be executed because a task having the higher priority is currently under execution.

(3) WAIT state

A task is placed in this state because it has issued a system call to request a resource. A task in this state can be an event flag waiting for an event, a semaphore waiting for a resource, a mail box waiting for a message, or waiting for time.

(4) DORMANT state

When resetting RX78K/IV, tasks other than the initial one are in a dormant state. When a task is terminated, it enters the pause state.

2.1.1 Task state transition

Fig. 2-1 illustrates state transition of a task. Each state transition has the following meaning, and system calls that places a task in one state from another are available.

(1) Starting

A task is started when it is placed in the READY state from the DORMANT state by a system call.

(2) Terminating

A task is terminated when it is placed in the DORMANT state from the RUN state by a system call.

(3) DISPATCH

Dispatching is to select a READY task to be placed in the RUN state. The task that will be placed in the RUN state has the highest priority of the tasks in the READY state. Dispatching is also called scheduling, and the block that executes dispatching or scheduling is called a dispatcher or scheduler.

There are 14 levels of task priorities (2 to 15), with the lowest number assigned the highest priority. Others include priorities 0 and 1. However, these cannot be specified for tasks because they are reserved for OS.

(4) PREEMPT

When a task having a higher priority than that of the task currently executed is placed in the RUN state, the currently executed task is once returned to the READY state. The task placed in the READY state from the RUN state cannot return to the RUN state, unless selected by the dispatcher again.

(5) Wait condition

If there is a necessity to wait for occurrence of an event, a task enters the WAIT state from the RUN state.

A task enters the WAIT state because the task has requested a resource while it was waiting for an event flag to be set but the requested reception has not been found, the task has requested reception of a message but the message has not arrived, the task has requested to wait for a fixed time period, but the period has not yet elapsed, etc.

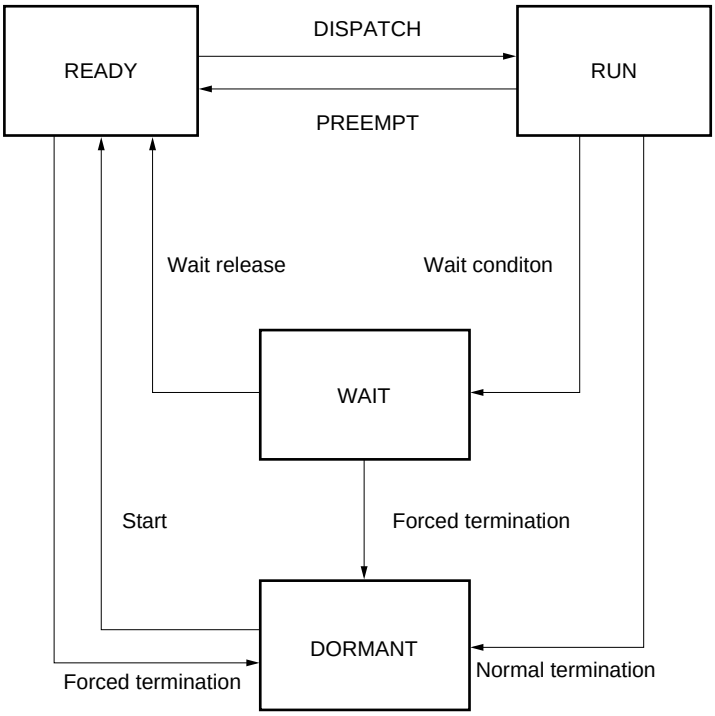
(6) Wait release

When a task is released from the WAIT state, it is placed in the READY state. In this case, an event is an occurrence that cancels the factor causing the WAIT state by the system.

(7) Forced termination

A task is forcibly terminated when it is placed in the DORMANT state from a state other than the RUN state by a system call.

Fig. 2-1 RX78K/IV Task State Transition



2.2 Task Manipulation

RX78K/IV allows the following manipulation of tasks:

(1) Starting task

All the tasks created when the system is started are placed in the DORMANT state.

A task in the DORMANT state cannot be dispatched until it is started. Therefore, it must be placed in the READY state by issuing system call “sta_tsk”.

(2) Terminating task

A task can be terminated by system call “ext_tsk” (normal termination) or “ter_tsk” (forced termination).

***** Interrupts are enabled when a task has been started for the first time. Disable interrupts for each task.

2.3 Synchronization and Communication

A task is executed under its own environments as an independent program. To execute two or more independent tasks in parallel and thereby to organize one system, it is necessary to control execution flow of the tasks, manage the resources shared by the tasks, and transfer information among tasks. Synchronization and communication are to implement these functions.

Synchronization is performed in two modes: wait and mutual exclusion.

The wait mode is used, when two tasks play the respective roles of work, to stop the work of one task until the other task has completed its work. In this case, the two tasks have a cooperative relationship. The mutual exclusion mode is used, when two tasks are using a common resource, to prevent one task from using the resource while the other task is using it. In this case, the two tasks are in a competitive relationship.

Communication between tasks is for transferring a message between tasks. A task that is to receive the message cannot proceed with processing until it has received the message. Therefore, the communication function can also be used in the place of the synchronization function.

As a means to implement synchronization and communication, RX78K/IV provides event flags and semaphores for synchronization, and mail boxes for communication.

2.3.1 Event flag

An event flag is used to make a task wait. Since an event flag is a 1-bit flag, it is not ORed or ANDed with the other flags.

A task issues system call “wai_flg”, “cwai_flg” or “pol_flg”, “cpol_flg” to wait for an occurrence of an event. If the event does not occur, the issuing task enters the WAIT state (when wai_flg or cwai_flg is issued). If the task does not enter the WAIT state, an error code is returned (when pol_flg or cpol_flg is issued).

To inform the occurrence of the event, system call “set_flg” is issued.

The event flag is cleared by system call “clr_flg”.

An event flag can be used to place two or more tasks in the WAIT state. Therefore, the event flag can queue tasks.

In this case, the tasks are released from the WAIT state when “set_flg” is issued once. If at least one task of the tasks placed in the WAIT state specifies clearing the flag (cwai_flg), the event flag is cleared after that task has been released from the WAIT state. Therefore, the tasks following that task are not released from the WAIT state.

2.3.2 Semaphore

A semaphore is used to mutually exclude tasks. The semaphore is a measurement that can check whether currently usable resources exist, or manage the number of resources.

The number of requested resources of the wait manipulation (P) and signal manipulation (V) instructions for a semaphore is fixed to 1.

If a task needs a resource, it performs resource acquisition manipulation (P instruction) by issuing system call “wai_sem”, “preq_sem”. If the number of resources is 1 or more, the number of resources of the semaphore is decremented by one.

If the number of resources is 0, the resource requesting task is placed in the WAIT state (wai_sem), or if the task is not placed in the WAIT state, an error code is returned (preq_sem). If the task does not need the resource any more, it executes resource return manipulation (V instruction) by issuing system call “sig_sem”. As a consequence, the number of resources of the semaphore is incremented by one. If there are tasks waiting for a resource at this time, the task first placed in the WAIT state is released. A task waits for a semaphore on a FIF0 basis only.

2.3.3 Mail box

A mail box transmits messages between tasks and has a queue of tasks waiting for reception of a message and a queue of messages waiting to be transmitted.

A task can request reception of a message from a mail box by issuing system call “rcv_msg” or “prcv_msg”. If the message has not arrived after “rcv_msg” has been issued, the task enters the WAIT state and queued to the mail box. If the task has issued “prcv_msg”, it does not enter the WAIT state, but an error code is returned. A task can also send a message to a mail box by issuing system call “snd_msg”. If there is a task waiting for the message, that task is released from the WAIT state and the message is transmitted to that task. If there is no waiting task, the message is queued. Messages and tasks are queued on a FIF0 basis only.

2.4 Interrupt Management

Interrupt management is for managing processing that is driven taking an interrupt as an event. A processing routine that is vectored by an interrupt is called an interrupt handler, which is independent of tasks. There are functions to set or exit execution from the interrupt handler.

To terminate the interrupt handler and pass control to a task, system calls “ret_int” and “ret_wup” are used. To issue “ret_int” from an interrupt handler occupying the register bank, select the register bank of the task executed before the interrupt has occurred, and then call “ret_int”.

Because RX78K/IV does not initialize each interrupt source, each source must be initialized by the user task.

Fig. 2-2 and 2-3 illustrate the flow of control when a task in the WAIT state is waken up by an event flag in an interrupt handler. If a task in the WAIT state is waken up an interrupt handler, system calls “ret_int” and “ret_wup” return control to the state of a task under execution when the interrupt has occurred, if the priority of the task to be waken up is lower than the priority of the task under execution when the interrupt has occurred.

If the task to be waken up has a higher priority than that of the task under execution, control is passed to the task to be waken up. Fig. 2-2 illustrates flow of control if task A has a higher priority than task B.

Caution
- **System calls cannot be issued during NMI interrupt processing.**
- **Please note that RX78K/IV does not support multiple interrupt among the interrupt handlers which are set for interrupt priority 3 (lowest level). Neither does it support them with the timer processing.**

<1> Task B has issued system call “wai_tsk” and is now in the WAIT state.
<2> Task A is under execution (has the highest priority of the tasks in the READY state).
<3> An interrupt occurs while task A is executed, and processing is passed to the interrupt handler.
<4> System call “iset_flg” is issued in the interrupt handler and task B is waken up.
<5> System call “ret_int” is issued in the interrupt handler.
<6> Because task A has a priority higher than that of task B, processing is returned to task A.

Fig. 2-2 Example of “ret_int”

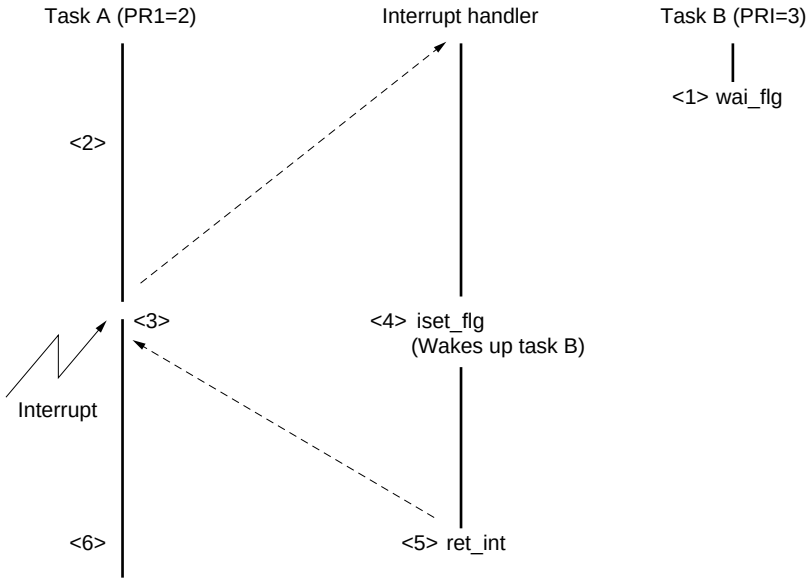
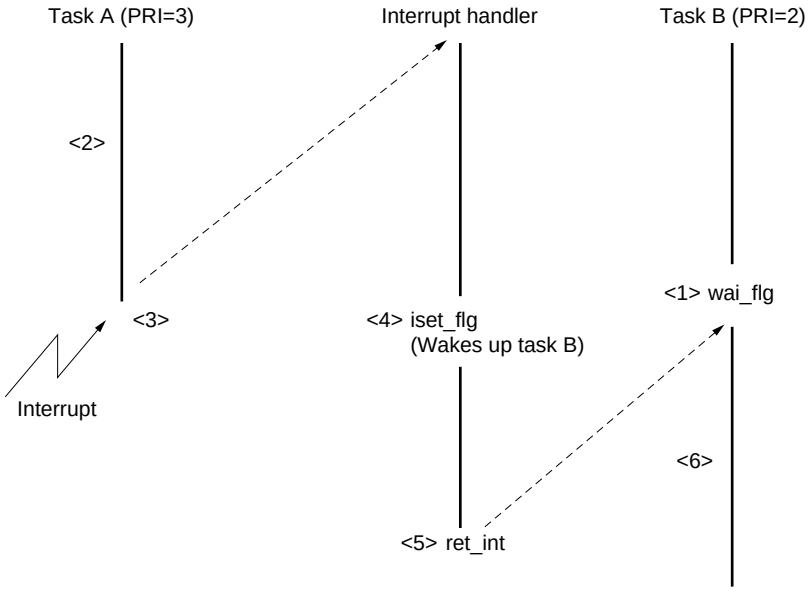


Fig. 2-3 shows an example where task A has a lower priority than task B.

<1> Task B has issued system call “wai_flg” and is now in the WAIT state.
<2> Task A is under execution (has the highest priority of the tasks in the READY state).
<3> An interrupt occurs while task A is executed, and processing is passed to the interrupt handler.
<4> System call “iset_flg” is issued in the interrupt handler, and task B is waken up.
<5> System call “ret_int” is issued in the interrupt handler.
<6> Because task B has a priority higher than task A, processing is passed to task B.

Fig. 2-3 Example of “ret_int”



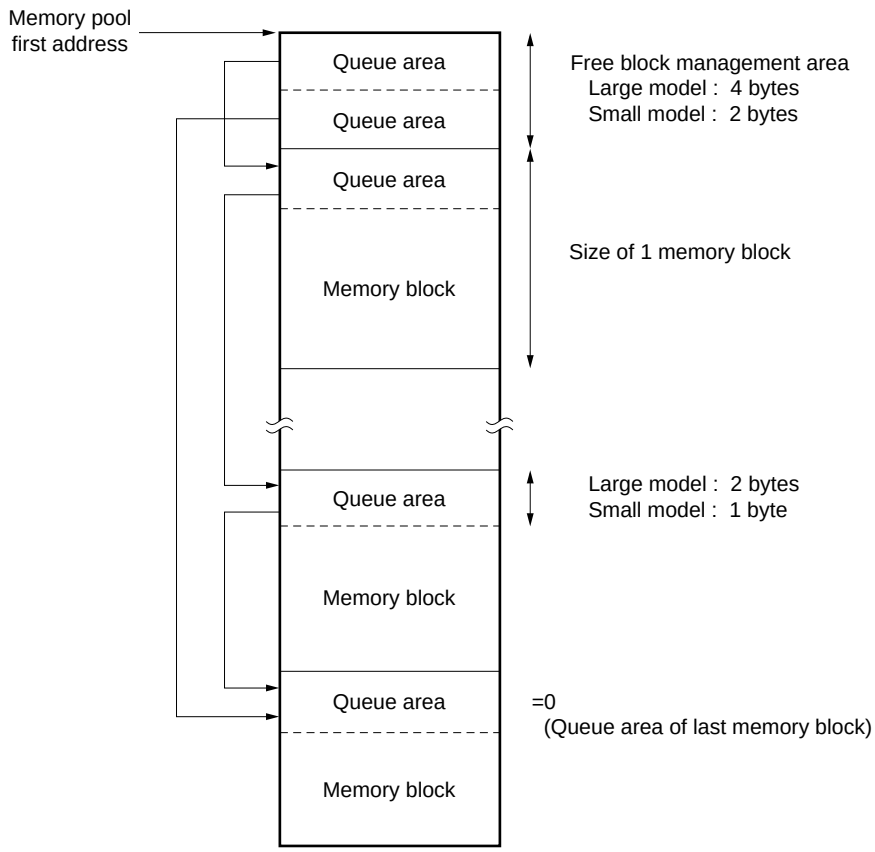
2.5 Memory Management

RX78K/IV performs dynamic memory management based on a memory block of fixed size. This means that a task acquires memory when necessary, and releases it when unnecessary. By using this feature, it is not necessary to acquire a memory area in advance for even two or more tasks, so that the memory can be efficiently used.

The memory block of RX78K/IV is dynamically acquired by a system call under execution by the system, and is returned after it has been used and is no longer necessary. If one memory block cannot be acquired, state transition of a task does not take place, but only an error code is returned. The size of one memory block is fixed to a value specified on system creation.

Usually, an acquired memory block is used as a message area, in which case the first 2 bytes with the large model, the first 1 byte with the small model, are used as a queue link area. Therefore, the area that can be used as a message area is the area excluding link area.

Fig. 2-4 Memory Pool Configuration



Caution When using a memory block as a message area, please make the size of the memory block a multiple of 2. Operation is not guaranteed if the size is not a multiple of 2.

2.6 Time management

Time management refers to delayed wake up of tasks and the running of cyclic handlers.

Delayed wake up tasks means performing wake up processing on a task after the elapsing of a specified amount of time, and is implemented by means of the “wai_task” system call. Running of cyclic handlers means managing the running of handlers as interrupt processes as specified time intervals, and is implemented by means of the “act_cyc(iact_cyc)” system call.

In RX78K/IV, one of the CPU’s 16-bit timer/counter units is used as the interval timer. By allocating vector table of the timer dispatch routine provided by the OS to this timer (?tmdsp), this routine can be periodically run. In the timer dispatch routine, the waiting time of a task that has issued the (wai_tsk) system call, or the waiting time of a cyclic handler is counted down, and when the specified time has elapsed, the task is awakened, or the cyclic handler is run.

Regarding the actual state transition, a task that has issued “wai_tsk” goes from the WAIT state to the READY state, whereas a cyclic handler is called directly from the timer dispatch routine.

[MEMO]

CHAPTER 3 RX78K/IV BASIC OPERATIONS AND CONTROL BLOCKS

This chapter describes the basic operations of a system running RX78K/IV and the control block configuration managed by RX78K/IV.

3.1 RX78K/IV Configuration

3.1.1 Flow of overall processing

RX78K/IV uses register bank 0 of the 8 register banks of the 78K/IV series microcomputers. For user tasks, multiple tasks can be allocated to register bank 1, and one task each to register bank 2 to 7.

The tasks allocated to register banks 2 to 7 can realize high-speed task switching, because with them, it is unnecessary to save the registers when tasks are switched over during execution.

An example method for allocating register banks is shown below.

Example 1: With 6 tasks
Register bank 1 : None
Register banks 2 to 7 : One task each

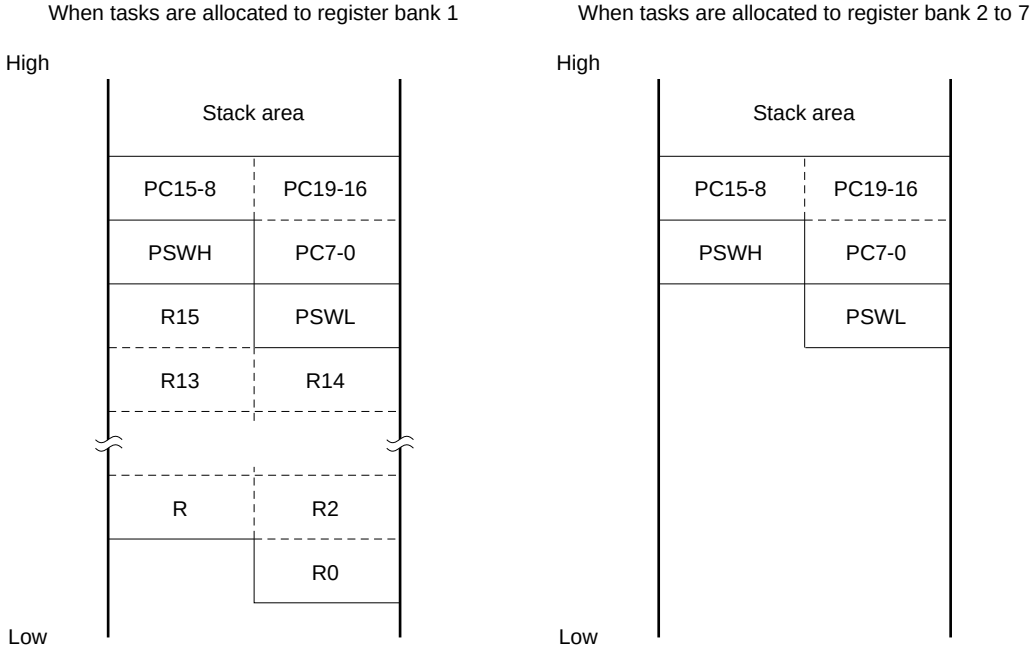
Example 2: With 6 tasks
Register bank 1 : 3 tasks
Register banks 2 to 4 : One task each
Register banks 5 to 7 : For interrupting

Example 3: With 10 tasks
Register bank 1 : 4 tasks
Register banks 2 to 7 : One task each

The task area is required for each task; therefore, when performing a task, the stack area of the task is used. The interrupt handler uses the stack area of the task being performed when an interrupt occurred.

The stack state in dispatching is shown below.

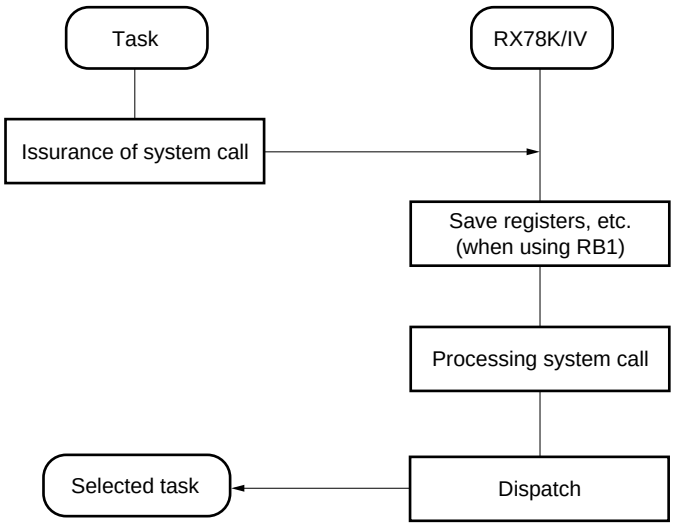
Fig. 3-1 Stack State in Dispatching



Register bank 1 is shared by multiple tasks. Therefore, dispatching or preempting make it necessary for registers to be saved or restored. This causes the overhead of these tasks to be slightly larger than the tasks allocated to register banks 2 to 7.

Fig. 3-2 shows the flow of the processing to be performed since the system call has been issued until control is returned to the bank.

Fig. 3-2 Flow of Overall Processing



3.1.2 Configuration of ready queue and dispatching

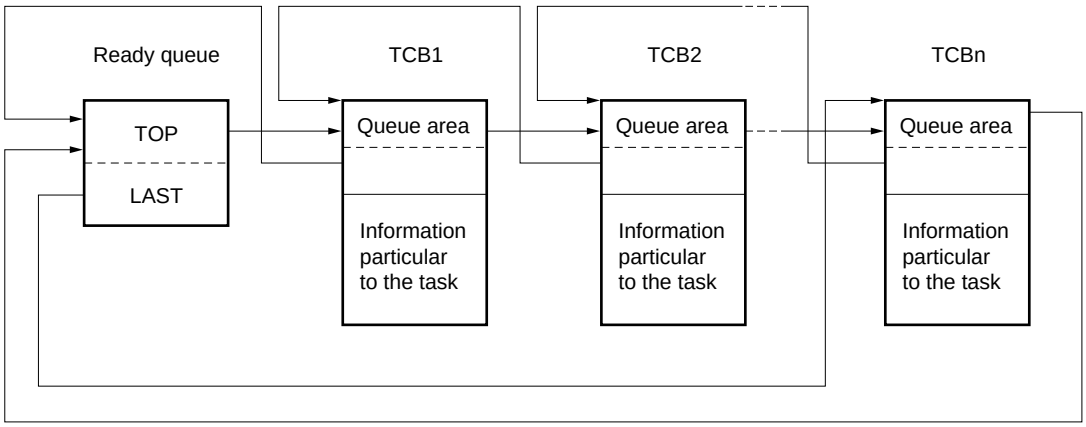
When RX78K/IV passes CPU execution privilege to a task, it searches for a task having the highest priority in the READY state. At this time, RX78K/IV references a ready queue.

RX78K/IV has 16 ready queues each corresponding to 16 priority levels (0 to 15). The task chained to the head of a ready queue has the highest priority of the tasks in the queue, and the task chained to the tail of the queue has the lowest priority. Among these priority levels, levels 2 to 15 can be specified for a task.

RX78K/IV searches the ready queues from the one having the highest priority for a task to be placed in the RUN state. However, if there is no executable user task at that time, it sets the CPU in the HALT mode. To resume execution of a user task, therefore, an interrupt must be used. The stack area at that time becomes the user stack area.

Fig. 3-3 shows the configuration of the ready queue.

Fig. 3-3 Ready Queue Configuration



3.2 RX78K/IV Control Block

RX78K/IV has control blocks to manage each object. The control blocks are created by the user at time of configuration, and RX78K/IV uses these blocks for system calls or for management. Therefore, the state of the control blocks can be always monitored by the user task. The types of RX78K/IV control blocks are as follows:

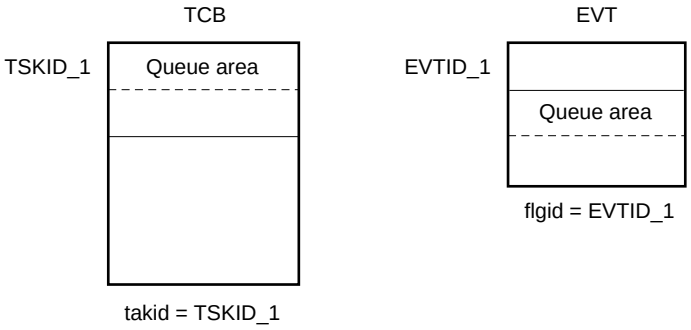
<1> TCB (Task Control Block)
<2> EVT (Eventflag)
<3> SEM (Semaphore)
<4> MBOX (Mail Box)
<5> MPL (Memory Pool)
<6> HCB (Cyclic Handler Control Block)

3.2.1 Control block memory area

As described above, the control blocks are created by the user at time of configuration with RX78K/IV. Therefore, all parameters are passed to system calls (related to control blocks) by specifying an address by using a symbol name.

Fig. 3-4 shows an example of parameters.

Fig. 3-4 Example of Parameters



3.2.2 Control blocks

(1) Task control block (TCB)

This control block is used by RX78K/IV to manage tasks. One control block of this type is assigned to each task. One TCB is 10 bytes long with the large model, and 8 bytes long with the small model.

(2) Event flag (EVT)

An event flag is a block that manages events that are used to synchronize tasks or make tasks wait.
One event flag is 4 bytes long with the large model, and 2 bytes long with the small model.

* If two or more tasks are waiting when the event flag is set, all the WAIT tasks enter the READY state. However, if a task exists that has issued “cwai_flg” and has entered the WAIT state, the flag is cleared after that task has been released from the WAIT state. After that, task state transitions do not take place.

(3) Semaphore (SEM)

A semaphore is a resource management block that is used to synchronize or mutually exclude tasks.
One semaphore is 4 bytes long with the large model, and 2 bytes long with the small model.

(4) Mail box (MBOX)

A mail box is a control block that manages message communication between tasks.
One mail box is 4 bytes long with the large model, and 2 bytes long with the small model.

(5) Memory pool (MPL)

The memory pool manages free memory blocks. Normally, memory blocks are used as message areas. When use it as a message area, be sure to set the size of one memory block to a multiple of 2. Even in cases other than messages, it is possible to secure only the portion of the memory blocks required for the relevant purpose.
The size of the memory pool is the total of the free block management area and the memory blocks.
The size of the free block management area is 4 bytes long with the large model, and 2 bytes long with the small model.

(6) Cyclic handler control block (HCB)

Cyclic handler control blocks are used by RX78K/IV to manage cyclic handlers. One cyclic handler is given one control block. The size of each cyclic handler control block is 10 bytes long with the large model, and 8 bytes long with the small model.

CHAPTER 4 RX78K/IV SYSTEM CALL LIST

This chapter describes the following items of each system calls:

[Function]:
Outline of the function of the system call

[Remarks]:
Describes the function of the system call

[System call ID number]:
Number particular to system call when the assembler calls the system call

[Parameter]:
Describes the name, data size, and explanation of each parameter.

[Return parameter]:
A list of return values that may occur as a result of issuance of the system call. The return value is returned as a function value of the system call in the case of the C language, and to register C in the case of the assembler. Since no abnormal termination processing is performed for all the system calls, add abnormal termination processing by referring to the value of the return parameter.

[Assembler format]:
Describes which register is used for output parameter, and how to handle an input parameter when the system call is issued in the assembler.
For the system call called from an interrupt handler, describes which register is used as an input parameter. The symbols described in the assembler format have the following addresses:

[Large model]
bnk0_b = 0FFEF3H, bnk0_vp = 0FFEF8H, bnk0_up = 0FFEFAH,
bnk0_e = 0FFEFCH, bnk0_d = 0FFEFDH

[Small model]
bnk0_b = 0FEF3H, bnk0_vp = 0FEF8H, bnk0_up = 0FEFAH,
bnk0_e = 0FEFCH

The system call which can be issued from an interrupt handler uses the register bank of that time.
The following is described in the assembler format:

[Large model]
Example E, UUP, VVP

[Small model]
Example E, UP, VP

[C format]:

Description format used when the system call is issued in the C language and the data type of the parameter

○ Interface with C language

The high-level language supported in developing an application program is the C language. To issue a system call in the C language, an assembler routine is necessary for interfacing with the OS. Therefore, the interface routine corresponding to the system call used must be linked with each task. The format of the system call in the C language is as follows:

ret = <name>([<parameter>],...);

ret	:	function return value (char type)
<name>	:	system call name
<parameter>	:	input parameter

The data type of each parameter and size are as follows.

char	:	8-bit integer
unsigned short	:	16-bit integer
char*	:	24-bit pointer (large model) 16-bit pointer (small model)

○ Interface with assembler

To issue a system call in the assembler, set parameters to the relevant register of register bank 0 as follows.

Example of sta_tsk issuance (large model)

```
MOV     BNK0_REG1, #system call ID number
MOVW    BNK0_REG2, #parameter 1 (lower 16 bits)
MOV     BNK0_REG3, #parameter 1 (higher 8 bits)
CALLT           [40H]
```

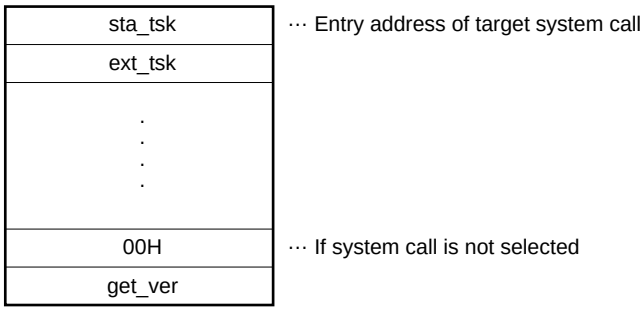
Example of sta_tsk issuance (small model)

```
MOV     BNK0_REG1, #system call ID number
MOVW    BNK0_REG2, #parameter 1
CALLT           [40H]
```

The entry address of branch processing to branch to the system call is set to 40H. The branch processing module takes out the entry address of each system call from a table created by the configurator and branches.

The configuration of the entry address storage table of system calls is as follows:

Fig. 4-1 Entry Address Storage Table



Be sure to directly call the following i×××_××× system call issued from the interrupt handler, without going through the branching module. The values in the brackets indicate the addresses of the CALLT table. However, addresses corresponding to unused system calls can be used by users.

ichg_pri (42H), irot_rdq (44H), iwup_tsk (46H), iset_flg (48H),
isig_sem (4AH), isnd_msg (4CH)

The ret_int and ret_wup system calls are branched out to system calls directly by the BR command.

○ Differences between “×××_×××” and “i×××_×××” system calls

System call “i×××_×××” is started by an interrupt handler and is not dispatched its own. This system call is dispatched when “ret_int” has been issued that returns execution from the interrupt handler, or “ret_wup” that returns execution from the interrupt handler and wakes up a task. When a system call is recovered from the interrupt handler with the RETI command, dispatching occurs in the next system call.

4.1 System Calls Related to Task

The following system calls are related to tasks:

(1)	sta_tsk	Starts task
(2)	ext_tsk	Normally terminates task itself
(3)	ter_tsk	Forcibly terminates another task
(4)	chg_pri	Changes the tasks priority
	ichg_pri	Changes the tasks priority
(5)	rot_rdq	Rotates specified priority ready queue
	irot_rdq	Rotates specified priority ready queue
(6)	tsk_sts	Checks the tasks state
(7)	slp_tsk	Places the tasks in WAIT state
(8)	wai_tsk	Places the tasks in WAIT state for fixed time
(9)	wup_tsk	Wakes up the task
	iwup_tsk	Wakes up the task (when used in interrupt handler)
(10)	can_wup	Makes the task wake-up request invalid

4.1.1 sta_tsk (Start Task)

[Function]

Starts a task by placing it in the READY state from the DORMANT state.

[Remarks]

This system call starts the task specified by tskid. As a result, the task enters the READY state from the DORMANT state. This system call must not be issued to a task that is not in the DORMANT state. If issued, error code “E_NODMT” is returned.

To start a task for the first time, or to restart a task that has been once terminated, processing is started from the start address of the task, and the value of the stack pointer is set to the initial value. The values of the other registers are undefined.

A task to which the “sta_tsk” system call is to be issued enables interrupts. Disable interrupts for each task.

[System call ID number]

sta_tsk = 0

[Parameter]

tskid (Task Identifier)	Task ID (TCB top address)	- 24 bits (large model) - 16 bits (small model)

[Return parameter]

E_OK	Normal termination
E_NODMT	Specified task is not in DORMANT state

[Assembler format]

[Large model]

```
MOV     bnk0_b, #0
MOVW    bnk0_up, #tskid
MOV     bnk0_d, #tskid
CALLT   [40H]
        :
```

C register= return parameter

[Small model]

```
MOV     bnk0_b, #0
MOVW    bnk0_up, #tskid
CALLT   [40H]
        :
```

C register= return parameter

[C format]

```
ret = sta_tsk(tskid);
char    *tskid;
```

4.1.2 ext_tsk (Exit Task)

[Function]

Normally terminates the task itself and places the task in the DORMANT state from the RUN state.

[Remarks]

This system call normally terminates the task that issues this system call itself.
As a result, the task enters the DORMANT state from the RUN state.
To restart by the “sta_tsk” system call the task that has entered the DORMANT state by issuing the “ext_tsk” system call, the start address and stack pointer are returned to the initial values. The wakeup request counter is cleared to 0.

The task does not automatically release the resource it has acquired before the “ext_tsk” system call is issued. Release the resources before issuing this system call.

[System call ID number]

ext_tsk = 1

[Parameter]

None

[Return parameter]

None (Control is not returned to the task that has issued this system call.)

[Assembler format]

[Large/Small model]

```
MOV     bnk0_b, #1
CALLT   [40H]
```

[C format]

```
ext_tsk ();
```

4.1.3 ter_tsk (Terminate Task)

[Function]

Forcibly aborts another task.

[Remarks]

This system call forcibly aborts the task specified by tskid.

If the specified task is placed in a queue, the task is released from the queue when the “ter_tsk” system call has been issued. Therefore, the task is released from the wait state before it is terminated.

“ter_tsk” does not automatically release the resources that the task has acquired before. Release the resources before terminating the task.

This system call cannot specify the task itself that issues it.

[System call ID number]

ter_tsk = 2

[Parameter]

tskid (Task Identifier)	Task ID (TCB top address)	- 24 bits (large model) - 16 bits (small model)

[Return parameter]

E_OK	Normal termination
E_DMT	Specified task is in DORMANT state

[Assembler format]

[Large model]

```
MOV     bnk0_b, #2
MOVW    bnk0_up, #tskid
MOV     bnk0_d, #tskid
CALLT   [40H]
        :
```

C register = return parameter

[Small model]

```
MOV     bnk0_b, #2
MOVW    bnk0_up, #tskid
CALLT   [40H]
        :
```

C register = return parameter

[C format]

```
ret = ter_tsk(tskid);
char    *tskid;
```

4.1.4 chg_pri (Change Task Priority)
ichg_pri (Change Task Priority for Interrupt)

[Function]

Changes the priority of a task.

[Remarks]

Changes the priority of the current task indicated by tskid to a value indicated by tskpri.
The own task is specified at tskid = 0. However, the own task cannot be specified with ichg_pri.
Priorities modified in the current system call are valid until remodified or performing the task is finished.
The priority for a task which has been restarted is the initial priority.

[System call ID number]

chg_pri = 3

[Parameter]

tskid (Task Identifier)	Task ID (TCB top address)	- 24 bits (large model) - 16 bits (small model)
tskpri(Task Priority)	Task priority	- 8 bits

[Return parameter]

E_OK	Normal termination
E_DMT	Specified task is in DORMANT state

[Assembler format]

- On issuing chg_pri

[Large model]

```
MOV     bnk0_b, #3
MOV     bnk0_e, #tskpri
MOVW    bnk0_up, #tskid
MOV     bnk0_d, #tskid
CALLT   [40H]
        :
```

C register = return parameter

[Small model]

```
MOV     bnk0_b, #3
MOV     bnk0_e, #tskpri
MOVW    bnk0_up, #tskid
CALLT   [40H]
        :
```

C register = return parameter

- On issuing ichg_pri

[Large model]

```
MOV     E, #tskpri
MOVG    UUP, #tskid
CALLT   [42H]
        :
```

C register = return parameter

[Small model]

```
MOV     E, #tskpri
MOVW    UP, #tskid
CALLT   [42H]
        :
```

C register = return parameter

[C format]

- On issuing chg-pri

```
ret = chg_pri (tskid, tskpri);
char    *tskid;
char    tskpri;
```

- On issuing ichg-pri

```
ret = ichg_pri (tskid, tskpri);
char    *tskid;
char    tskpri;
```

4.1.5 rot_rdq (Rotate Ready Queue)
irot_rdq (Rotate Ready Queue for Interrupt)

[Function]

Rotates the ready queue of the specified priority.

[Remarks]

Rotates the ready queue of the priority indicated by tskpri. Consequently, the task placed at the top of the ready queue is placed at the end of the queue and execution of the task of the same priority is changed.

When tskpri = 0, the ready queue of the priority of the task itself is rotated. Therefore, this system call can be issued to relinquish execution privileges so that the task itself is placed at the end of the ready queue. However, irot_rdq cannot specify tskpri = 0.

If no task exists in the ready queue of the specified priority, nothing is executed, but this does not result in an error.

[System call ID number]

rot_rdq = 4

[Parameter]

tskpri (Task Priority)	Task priority	- 8 bits

[Return parameter]

E_OK	Normal termination

[Assembler format]

- On issuing rot_rdq

```
[Large/Small model]
MOV     bnk0_v, #4
MOV     bnk0_e, #tskpri
CALLT   [40H]
        :
```

C register = return parameter

- On issuing irot_rdq

```
[Large/Small model]
MOV     E, #tskpri
CALLT   [44H]
        :
```

C register = return parameter

[C format]

- On issuing rot_rdq

```
ret = rot_rdq (tskpri);
char    tskpri;
```

- On issuing irot_rdq

```
ret = irot_rdq (tskpri);
char    tskpri;
```

4.1.6 tsk_sts (Get Task Status)

[Function]

Gets a task's state.

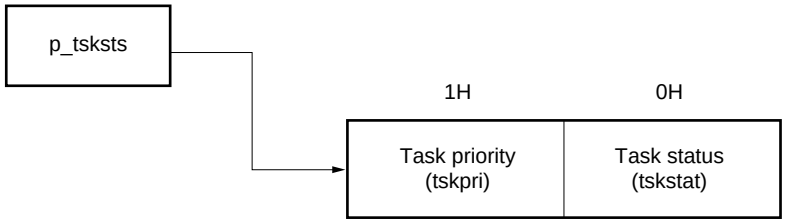
[Remarks]

References each status of the task indicated by tskid, and sets the current priority of the target task "tskpri" and task state "tskstat" in a specified address (refer to the figure below).

The own task is specified at tskid = 0.

The area for P-tsksts should be allocated by the user.

For the task status, refer to **APPENDIX 1**.



[System call ID number]

tsk_sts = 5

[Parameter]

tskid (Task Identifier)	Task ID (TCB top address)	- 24 bits (large model) - 16 bits (small model)
p_tsksts (Pointer of Task Status)	Address of area storing task status	- 24 bits (large model) - 16 bits (small model)

[Return parameter]

E_OK	Normal termination

[Assembler format]

[Large model]

```
MOV     bnk0_b, #5
MOVW    bnk0_up, #tskid
MOV     bnk0_d, #tskid
MOVW    bnk0_vp, #p_tsksts
MOV     bnk0_e, #p_tsksts
CALLT   [40H]
        :
```

C register = return parameter

[Small model]

```
MOV     bnk0_b, #5
MOVW    bnk0_up, #tskid
MOVW    bnk0_vp, #p_tsksts
CALLT   [40H]
        :
```

C register = return parameter

[C format]

```
ret = tsk_sts (p_tsksts,tskid);
short       *p_tsksts;
char        *tskid;
```

4.1.7 slp_tsk (Sleep Task)

[Function]

Places a task in the WAIT state.

[Remarks]

Places the task itself in the WAIT state from the RUN state. This WAIT state is released issuing the system call “wup_tsk (iwup_tsk)” or “ret_wup” to the task.

[System call ID number]

slp_tsk = 6

[Parameter]

None

[Return parameter]

E_OK	Normal termination

[Assembler format]

[Large/Small model]

```
MOV     bnk0_b, #6
CALLT   [40H]
        :
```

C register = return parameter

[C format]

```
ret = slp_tsk ();
```

4.1.8 wai_tsk (Wait for Wakeup Task)

[Function]

Places a task in the WAIT state for a fixed amount of time.

[Remarks]

Places the task itself in the WAIT state for a fixed amount of time from the RUN state. This WAIT state can be released issuing the system call “wup_tsk (iwup_tsk)” or “ret_wup” to this task, or when the amount of time specified by tmout has elapsed. If the value of tmout is 0, immediate return is executed, and E_TMOUT is returned as the return parameter. If tmout is 0 and the number of wake-up request counts is 1 or more, E_OK is returned (and the wake-up request count is updated).

When this system call is used, the vector address of the timer to be used must be set. For the timer to be used, refer to **5.1 Hardware Used**.

Wait time = Set interrupt time × TMOUT value

[System call ID number]

wai_tsk = 7

[Parameter]

tmout (Timeout)	Time out specification	- 16 bits

[Return parameter]

E_OK	Normal termination
E_TMOUT	Elapsed time

[Assembler format]

[Large/Small model]

```
MOV     bnk0_b, #7
MOVW    bnk0_vp, #tmout
CALLT   [40H]
        :
```

C register = return parameter

[C format]

```
ret = wai_tsk (tmout);
unsigned short      tmout;
```

4.1.9 wup_tsk (Wakeup Task)
iwup_tsk (Wakeup Task for Interrupt)

[Function]

Wakes up a task and places it in the READY state from the WAIT state.

[Remarks]

Places a task that has been placed in the WAIT state by system call “slp_tsk” or “wai_tsk” in the READY state. The target task is indicated by tskid. The task itself cannot be specified.

If the target task has not executed “slp_tsk” or “wai_tsk” and is not in the WAIT state, this “wup_tsk” request is queued. In this case, this “wup_tsk” request becomes valid when the target task later executes “slp_tsk” or “wai_tsk”.

If the number of wake-up requests in the queue is about to exceed the upper limit (0FFH), an error (E_QOVR) is returned.

[System call ID number]

wup_tsk = 8

[Parameter]

tskid (Task Identifier)	Task ID (TCB top address)	- 24 bits (large model) - 16 bits (small model)

[Return parameter]

E_OK	Normal termination
E_DMT	Specified task is in DORMANT state
E_QOVR	Queue overflow

[Assembler format]

- On issuing wup_tsk

[Large model]

```
MOV     bnk0_b, #8
MOVW    bnk0_up, #tskid
MOV     bnk0_d, #tskid
CALLT   [40H]
        :
C register = return parameter
```

[Small model]

```
MOV     bnk0_b, #8
MOVW    bnk0_up, #tskid
CALLT   [40H]
        :
C register = return parameter
```

- On issuing iwup_tsk

[Large model]

```
MOVG    UUP, #tskid
CALLT   [46H]
        :
C register = return parameter
```

[Small model]

```
MOVW    UP, #tskid
CALLT   [46H]
        :
C register = return parameter
```

[C format]

- On issuing wup_tsk

```
ret = wup_tsk (tskid);
char    *tskid;
```

- On issuing iwup_tsk

```
ret = iwup_tsk (tskid);
char    *tskid;
```

4.1.10 can_wup (Cancel Wakeup Task)

[Function]

Cancels a task wake-up request.

[Remarks]

Stores the number of wake-up requests queued to the target task indicated by tskid to a specified pointer, and at the same time cancels all wake-up requests. The area for p_wupcnt should be allocated by the user.
The own task is specified at tskid = 0.

[System call ID number]

can_wup = 9

[Parameter]

tskid (Task Identifier)	Task ID (TCB top address)	- 24 bits (large model) - 16 bits (small model)
p_wupcnt (Pointer of Wakeup Count)	Pointer storing the number of wake-up requests queued	- 24 bits (large model) - 16 bits (small model)

[Return parameter]

E_OK	Normal termination
E_DMT	Specified task is in DORMANT state

[Assembler format]

[Large model]

```
MOV     bnk0_b, #9
MOVW    bnk0_up, #tskid
MOV     bnk0_d, #tskid
MOVW    bnk0_vp, #p_wupcnt
MOV     bnk0_e, #p_wupcnt
CALLT   [40H]
        :
```

C register = return parameter

[Small model]

```
MOV     bnk0_b, #9
MOVW    bnk0_up, #tskid
MOVW    bnk0_vp, #p_wupcnt
CALLT   [40H]
        :
```

C register = return parameter

[C format]

```
ret = can_wup (p_wupcnt,tskid);
char    *p_wupcnt;
char    *tskid;
```

4.2 Synchronization and Communication

The following system calls are used for synchronization and communication:

(1)	set_flg	Sets an event flag
(2)	iset_flg	Sets an event flag (when used in interrupt handler)
(3)	clr_flg	Clears an event flag
(4)	wai_flg	Waits for an event flag (without clear)
(5)	cwai_flg	Waits for an event flag (with clear)
(6)	pol_flg	Obtains an event flag (without clear)
(7)	cpol_flg	Obtains an event flag (with clear)
(8)	sig_sem	Signal manipulation of a semaphore (V instruction)
(9)	isig_sem	Signal manipulation of a semaphore (when used in interrupt handler)
(10)	wai_sem	Wait manipulation of a semaphore (P instruction)
(11)	preq_sem	Obtains the semaphore resource
(12)	snd_msg	Sends a message
(13)	isnd_msg	Sends a message (when used in interrupt handler)
(14)	rcv_msg	Waits for message reception
(15)	prcv_msg	Receives a message

4.2.1 set_flg (Set Eventflag)
iset_flg (Set Eventflag for Interrupt)

[Function]

Sets an event flag.

[Remarks]

The event flag indicated by flgid is set to 1. If a task exists that has been made to wait for the event flag by “wai_flg, cwai_flg”, the task is released from the WAIT state and placed in the READY state.

Two or more tasks can wait for the same event flag. In this case, the tasks can be released from the WAIT state when “set_flg” is issued once.

[System call ID number]

set_flg = 10

[Parameter]

flgid (Eventflag Identifier)	Event flag ID (Eventflag top address)	- 24 bits (large model) - 16 bits (small model)

[Return parameter]

E_OK	Normal termination

[Assembler format]

- On issuing set_flg

[Large model]

```
MOV     bnk0_b, #10
MOVW    bnk0_up, #flgid
MOV     bnk0_d, #flgid
CALLT   [40H]
        :
```

C register = return parameter

[Small model]

```
MOV     bnk0_b, #10
MOVW    bnk0_up, #flgid
CALLT   [40H]
        :
```

C register = return parameter

- On issuing iset_flg

[Large model]

```
MOVG    UUP, #flgid
CALLT   [48H]
        :
```

C register = return parameter

[Small model]

```
MOVW    UP, #flgid
CALLT   [48H]
        :
```

C register = return parameter

[C format]

- On issuing set_flg

```
ret = set_flg (flgid);
char    *flgid;
```

- On issuing iset_flg

```
ret = iset_flg (flgid);
char    *flgid;
```

4.2.2 clr_flg (Clear Eventflag)

[Function]

Clears an event flag.

[Remarks]

Clears the event flag indicated by flgid. Even if a task exists that has been made to wait for the event flag by “wai_flg”, “cwai_flg”, that task is not released from the WAIT state.

[System call ID number]

clr_flg = 11

[Parameter]

flgid (Eventflag Identifier)	Event flag ID (Eventflag top address)	- 24 bits (large model) - 16 bits (small model)

[Return parameter]

E_OK	Normal termination

[Assembler format]

[Large model]

```
MOV     bnk0_b, #11
MOVW    bnk0_up, #flgid
MOV     bnk0_d, #flgid
CALLT   [40H]
        :
```

C register = return parameter

[Small model]

```
MOV     bnk0_b, #11
MOVW    bnk0_up, #flgid
CALLT   [40H]
        :
```

C register = return parameter

[C format]

```
ret = clr_flg (flgid);
char    *flgid;
```

4.2.3 wai_flg (Wait Eventflag)

[Function]

Waits for an event flag (without clear).

[Remarks]

Waits until an event flag indicated by flgid is set to 1. The value of the event flag remains set after it has been set and the task has been released from the WAIT state.

Two or more tasks can wait for the same event flag.

[System call ID number]

wai_flg = 12

[Parameter]

flgid (Eventflag Identifier)	Event flag ID (Eventflag top address)	- 24 bits (large model) - 16 bits (small model)

[Return parameter]

E_OK	Normal termination

[Assembler format]

[Large model]

```
MOV     bnk0_b, #12
MOVW    bnk0_up, #flgid
MOV     bnk0_d, #flgid
CALLT   [40H]
        :
```

C register = return parameter

[Small model]

```
MOV     bnk0_b, #12
MOVW    bnk0_up, #flgid
CALLT   [40H]
        :
```

C register = return parameter

[C format]

```
ret = wai_flg (flgid);
char    *flgid;
```

4.2.4 cwai_flg (Wait and Clear Eventflag)

[Function]

Waits for an event flag (with clear).

[Remarks]

Waits until an event flag indicated by flgid is set to 1. The event flag is cleared when it has been set and the task has been released from the WAIT state.

If “set_flg” is issued when two or more tasks are waiting for an event flag, the tasks are sequentially woken up starting from the one placed at the top of the queue. However, the tasks following the one in the WAIT state by “cwai_flg” cannot be woken up because the event flag is cleared.

[System call ID number]

cwai_flg = 13

[Parameter]

flgid (Eventflag Identifier)	Event flag ID (Eventflag top address)	- 24 bits (large model) - 16 bits (small model)

[Return parameter]

E_OK	Normal termination

[Assembler format]

[Large model]

```
MOV     bnk0_b, #13
MOVW    bnk0_up, #flgid
MOV     bnk0_d, #flgid
CALLT   [40H]
        :
```

C register = return parameter

[Small model]

```
MOV     bnk0_b, #13
MOVW    bnk0_up, #flgid
CALLT   [40H]
        :
```

C register = return parameter

[C format]

```
ret = cwai_flg (flgid);
char    *flgid;
```

4.2.5 pol_flg (Poll Eventflag)

[Function]

Obtains an event flag (without clear).

[Remarks]

Checks whether the event flag indicated by flgid is set. If the flag is set, the task is terminated normally; if not, an error is returned (E_PLFAIL). In either case, the original value of the event flag is retained.

[System call ID number]

pol_flg = 14

[Parameter]

flgid (Eventflag Identifier)	Event flag ID (Eventflag top address)	- 24 bits (large model) - 16 bits (small model)

[Return parameter]

E_OK	Normal termination
E_PLFAIL	Polling failed

[Assembler format]

[Large model]

```
MOV     bnk0_b, #14
MOVW    bnk0_up, #flgid
MOV     bnk0_d, #flgid
CALLT   [40H]
        :
```

C register = return parameter

[Small model]

```
MOV     bnk0_b, #14
MOVW    bnk0_up, #flgid
CALLT   [40H]
        :
```

C register = return parameter

[C format]

```
ret = pol_flg (flgid);
char    *flgid;
```

4.2.6 cpol_flg (Poll and Clear Eventflag)

[Function]

Obtains an event flag (with clear).

[Remarks]

Checks whether the event flag indicated by flgid is set. If the flag is set, the flag is cleared and the task is terminated normally; if not, an error is returned (E_PLFAIL) with the flag value left untouched.

[System call ID number]

cpol_flg = 15

[Parameter]

flgid (Eventflag Identifier)	Event flag ID (Eventflag top address)	- 24 bits (large model) - 16 bits (small model)

[Return parameter]

E_OK	Normal termination	- 8 bits
E_PLFAIL	Polling failed	- 8 bits

[Assembler format]

[Large model]

```
MOV     bnk0_b, #15
MOVW    bnk0_up, #flgid
MOV     bnk0_d, #flgid
CALLT   [40H]
        :
```

C register = return parameter

[Small model]

```
MOV     bnk0_b, #15
MOVW    bnk0_up, #flgid
CALLT   [40H]
        :
```

C register = return parameter

[C format]

```
ret = cpol_flg (flgid);
char    *flgid;
```

4.2.7 sig_sem (Signal Semaphore)
isig_sem (Signal Semaphore for Interrupt)

[Function]

Executes signal manipulation for semaphore (V instruction).

[Remarks]

If tasks exist that wait for a semaphore indicated by semid, the task at the top of the queue is placed in the READY state. If no task is waiting for the semaphore, the count value of the semaphore is increased by one.
If the count value of the semaphore is about to exceed the upper limit (0FFH), “E_QOVR” is returned as an error code.

[System call ID number]

sig_sem = 16

[Parameter]

semid (Semaphore Identifier)	Semaphore ID (Semaphore top address)	- 24 bits (large model) - 16 bits (small model)

[Return parameter]

E_OK	Normal termination
E_QOVR	Queue overflow

[Assembler format]

- On issuing sig_sem

[Large model]

```
MOV     bnk0_b, #16
MOVW    bnk0_up, #semid
MOV     bnk0_d, #semid
CALLT   [40H]
        :
```

C register = return parameter

[Small model]

```
MOV     bnk0_b, #16
MOVW    bnk0_up, #semid
CALLT   [40H]
        :
```

C register = return parameter

- On issuing isig_sem

[Large model]

```
MOVG    UUP, #semid
CALLT   [4AH]
        :
```

C register = return parameter

[Small model]

```
MOVW    UP, #semid
CALLT   [4AH]
        :
```

C register = return parameter

[C format]

- On issuing sig_sem

```
ret = sig_sem (semid);
char    *semid;
```

- On issuing isig_sem

```
ret = isig_sem (semid);
char    *semid;
```

4.2.8 wai_sem (Wait on Semaphore)

[Function]

Executes wait manipulation for a semaphore (P instruction).

[Remarks]

If the count value of the semaphore indicated by semid is 1 or more, the count value of the semaphore is decremented by one, and to the task that issued this system call. If the count value of the semaphore is 0, the count value of the semaphore is not changed, and the task that has issued this system call is placed in the WAIT state.

[System call ID number]

wai_sem = 17

[Parameter]

semid (Semaphore Identifier)	Semaphore ID (Semaphore top address)	- 24 bits (large model) - 16 bits (small model)

[Return parameter]

E_OK	Normal termination

[Assembler format]

[Large model]

```
MOV     bnk0_b, #17
MOVW    bnk0_up, #semid
MOV     bnk0_d, #semid
CALLT   [40H]
        :
```

C register = return parameter

[Small model]

```
MOV     bnk0_b, #17
MOVW    bnk0_up, #semid
CALLT   [40H]
        :
```

C register = return parameter

[C format]

```
ret = wai_sem(semid);
char    *semid;
```

4.2.9 preq_sem (Poll and Request Semaphore)

[Function]

Obtains a semaphore resource.

[Remarks]

If the count value of the semaphore indicated by semid is 1 or more, the count value of the semaphore is decremented by one, and the task terminates normally. If the count value of the semaphore is 0, the semaphore value is not changed and an error is returned (E_PLFAIL).

[System call ID number]

preq_sem = 18

[Parameter]

semid (Semaphore Identifier)	Semaphore ID (Semaphore top address)	- 24 bits (large model) - 16 bits (small model)

[Return parameter]

E_OK	Normal termination
E_PLFAIL	Polling failed

[Assembler format]

[Large model]

```
MOV     bnk0_b, #18
MOVW    bnk0_up, #semid
MOV     bnk0_d, #semid
CALLT   [40H]
        :
```

C register = return parameter

[Small model]

```
MOV     bnk0_b, #18
MOVW    bnk0_up, #semid
CALLT   [40H]
        :
```

C register = return parameter

[C format]

```
ret = preq_sem(semid);
char    *semid;
```

4.2.10 snd_msg (Send Message to Mailbox)
isnd_msg (Send Message to Mailbox for Interrupt)

[Function]

Sends a message.

[Remarks]

Sends a message at the address pk_msg to a mail box indicated by mbxid. If a task exists that is waiting for the message, the top address of the sent message is returned, and the task is placed in the READY state. In this case, the message is not sent to the mail box.

If there is no task waiting for the message, the task issuing “snd_msg” is not placed in the WAIT state. This system call only sends a message to a mail box, and execution of the task continues. In other words, message transmission is carried out asynchronously.

The first 2 bytes of the transmitting message are used as the link area of the queue. Therefore, be sure to store the message from the 3rd byte and onward.

[System call ID number]

snd_msg = 19

[Parameter]

mbxid (Mailbox Identifier)	Mail box ID (Mailbox top address)	- 24 bits (large model) - 16 bits (small model)
pk_msg (Message Packet)	Transmission message top address	- 24 bits (large model) - 16 bits (small model)

[Return parameter]

E_OK	Normal termination

[Assembler format]

- On issuing snd_msg

[Large model]

```
MOV     bnk0_b, #19
MOVW    bnk0_up, #mbxid
MOV     bnk0_d, #mbxid
MOVW    bnk0_vp, #pk_msg
MOV     bnk0_e, #pk_msg
CALLT   [40H]
        :
```

C register = return parameter

[Small model]

```
MOV     bnk0_b, #19
MOVW    bnk0_up, #mbxid
MOVW    bnk0_vp, #pk_msg
CALLT   [40H]
        :
```

C register = return parameter

- On issuing isnd_msg

[Large model]

```
MOVG    UUP, #mbxid
MOVG    VVP, #pk_msg
CALLT   [4CH]
        :
```

C register = return parameter

[Small model]

```
MOVW    UP, #mbxid
MOVW    VP, #pk_msg
CALLT   [4CH]
        :
```

C register = return parameter

[C format]

- On issuing snd_msg

```
ret = snd_msg (mbxid, pk_msg);
char        *mbxid;
char        *pk_msg;
```

- On issuing isnd_msg

```
ret = isnd_msg (mbxid, pk_msg);
char        *mbxid;
char        *pk_msg;
```

4.2.11 rcv_msg (Receive Message from Mailbox)

[Function]

Waits for reception from a mail box.

[Remarks]

Receives a message from a mail box indicated by mbxid and sets the top address of the received message in an address specified by the parameter. If the message has not arrived at the mail box, the task that has issued this system call is placed in the WAIT state. The area for ppk-msg should be allocated by the user. Received message becomes a top address of the message area actual data is stored.

[System call ID number]

rcv_msg = 20

[Parameter]

mbxid (Mailbox Identifier)	Mail box ID (Mailbox top address)	- 24 bits (large model) - 16 bits (small model)
ppk_msg (Pointer of Message Packet)	Address of area storing the top address of received message	- 24 bits (large model) - 16 bits (small model)

[Return parameter]

E_OK	Normal termination

[Assembler format]

[Large model]

```
MOV     bnk0_b, #20
MOVW    bnk0_up, #mbxid
MOV     bnk0_d, #mbxid
MOVG    VVP, #ppk_msg
PUSH    VVP
CALLT   [40H]
        :
```

C register = return parameter

[Small model]

```
MOV     bnk0_b, #20
MOVW    bnk0_up, #mbxid
MOVW    AX, #ppk_msg
PUSH    AX
CALLT   [40H]
        :
```

C register = return parameter

[C format]

```
ret = rcv_msg (ppk_msg,mbxid);
char    **ppk_msg;
char    *mbxid;
```

4.2.12 prcv_msg (Poll and Receive Message from Mailbox)

[Function]

Receives a message.

[Remarks]

If a message has arrived to the mail box indicated by mbxid, it receives the message and sets the top address of the received message in an address specified by the parameter. In this case, this system call terminates normally. If the message has not arrived to the mail box, this system call returns “E_PLFAIL” as an error code. The area for ppk-msg should be allocated by the user.

Received message becomes a top address of the message area actual data is stored.

[System call ID number]

prcv_msg = 21

[Parameter]

mbxid (Mailbox Identifier)	Mail box ID (Mailbox top address)	- 24 bits (large model) - 16 bits (small model)
ppk_msg (Pointer of Message Packet)	Address of area storing the top address of received message	- 24 bits (large model) - 16 bits (small model)

[Return parameter]

E_OK	Normal termination
E_PLFAIL	Polling failed

[Assembler format]

[Large model]

```
MOV     bnk0_b, #21
MOVW    bnk0_up, #mbxid
MOV     bnk0_d, #mbxid
MOVW    bnk0_vp, #ppk_msg
MOV     bnk0_e, #ppk_msg
CALLT   [40H]
        :
```

C register = return parameter

[Small model]

```
MOV     bnk0_b, #21
MOVW    bnk0_up, #mbxid
MOVW    bnk0_vp, #ppk_msg
CALLT   [40H]
        :
```

C register = return parameter

[C format]

```
ret = prcv_msg (ppk_msg,mbxid);
char    **ppk_msg;
char    *mbxid;
```

4.3 Memory Management

The following system calls are related to memory:

(1)	pget_blk	Gets a fixed-length memory block
(2)	rel_blk	Releases a fixed-length memory block

4.3.1 pget_blk (Poll and Get Fixed-Length Memory Block)

[Function]

Gets a memory block of a fixed length.

[Remarks]

Gets a memory block of a fixed length from a memory pool indicated by mplid. This memory block can be used as a message area for communication between tasks. The size of the memory block is determined on creation of the system and is fixed. When a memory block can be gotten, the top address of the memory block is set in an address specified by the parameter. If the memory block cannot be gotten, an error code "E_PLFAIL" is returned. In either case, the task is not placed in the WAIT state.

The area for p-blk should be allocated by the user.

[System call ID number]

pget_blk = 22

[Parameter]

mplid (Memory Pool Identifier)	Memory pool ID	- 24 bits (large model)
	(Memory pool top address)	- 16 bits (small model)
p_blk (Pointer of Block Start Address)	Address of area storing the top	- 24 bits (large model)
	address of the memory block	- 16 bits (small model)

[Return parameter]

E_OK	Normal termination
E_PLFAIL	Polling failed

[Assembler format]

[Large model]

```
MOV     bnk0_b, #22
MOVW    bnk0_up, #mplid
MOV     bnk0_d, #mplid
MOVW    bnk0_vp, #p_blk
MOV     bnk0_e, #p_blk
CALLT   [40H]
        :
```

C register = return parameter

[Small model]

```
MOV     bnk0_b, #22
MOVW    bnk0_up, #mplid
MOVW    bnk0_vp, #p_blk
CALLT   [40H]
        :
```

C register = return parameter

[C format]

```
ret = pget_blk (p_blk,mplid);
char    **p_blk;
char    *mplid;
```

4.3.2 rel_blk (Release Fixed-Length Memory Block)

[Function]

Releases a memory block of a fixed length.

[Remarks]

Releases the memory block indicated by blk to a memory pool specified by mplid.

[System call ID number]

rel_blk = 23

[Parameter]

mplid (Memory Pool Identifier)	Memory pool ID	- 24 bits (large model)
	(Memory pool top address)	- 16 bits (small model)
blk (Block Start Address)	Top address of the memory block	- 24 bits (large model)
		- 16 bits (small model)

[Return parameter]

E_OK	Normal termination

[Assembler format]

[Large model]

```
MOV     bnk0_b, #23
MOVW    bnk0_up, #mplid
MOV     bnk0_d, #mplid
MOVW    bnk0_vp, #blk
MOV     bnk0_e, #blk
CALLT   [40H]
        :
```

C register = return parameter

[Small model]

```
MOV     bnk0_b, #23
MOVW    bnk0_up, #mplid
MOVW    bnk0_vp, #blk
CALLT   [40H]
        :
```

C register = return parameter

[C format]

```
ret = rel_blk (mplid,blk);
char    *mplid;
char    *blk;
```

4.4 Interrupt Processing

The following system calls can be used for interrupt processing:

(1)	ret_int	Terminates the interrupt handler
(2)	ret_wup	Returns from an interrupt and wakes up a task

4.4.1 ret_int (Return from Interrupt Handler)

[Function]

Returns from an interrupt handler.

[Remarks]

This system call is issued when execution is returned from an interrupt handler.

Even if “i×××_yyy” system call is issued in the interrupt handler, dispatching does not take place and is delayed until execution exits from the interrupt handler after this system call has been issued.

[System call ID number]

None

[Parameter]

None

[Return parameter]

None.

Does not return to the interrupt handler that issued the system call.

[Assembler format]

[Large/Small model]

```
BR          !ret_int
```

Unlike the other system calls, this system call is caused to branch by the jump instruction.

Caution When this system call is issued from an interrupt handler, be sure to save the register values before the interrupt occured to the stack. Save the register values to the stack in the sequence shown in Fig. 4-2. If the interrupt handler occupies a register bank, select the register bank of the task executed before the interrupt has occurred, and then call this system call.

[C format]

```
ret_int ();
```

Remark Because this system call is only supported in version 2.00 or above of the C complier CC78K4, it cannot be used with a C complier of a version lower than 2.00. With a C complier of a version 2.00 or above, specify “#pragma∆rtos_interrupt” (refer to **APPENDIX 2**).

4.4.2 ret_wup (Return and Wakeup Task)

[Function]

Returns from an interrupt handler and wakes up a task.

[Remarks]

Returns from an interrupt handler and wakes up an interrupt processing task indicated by tskid (the task placed in the WAIT state by “slp_tsk” or “wai_tsk”). If the target task does not execute “slp_tsk” or “wai_tsk”, the request is queued.

Even if “i×××_yyy” system call is issued in the interrupt handler, dispatching does not take place and is delayed until execution exits from the interrupt handler after this system call has been issued.

[System call ID number]

None

[Parameter]

tskid (Task Identifier)	Task ID (TCB top address)	- 24 bits (large model) - 16 bits (small model)

[Return parameter]

None.

Does not return to the interrupt handler that issued the system call.

[Assembler format]

[Large/Small model]

```
MOVG    UUP, #tskid
BR      !ret_wup
```

Unlike the other system calls, this system call is caused to branch by the jump instruction.

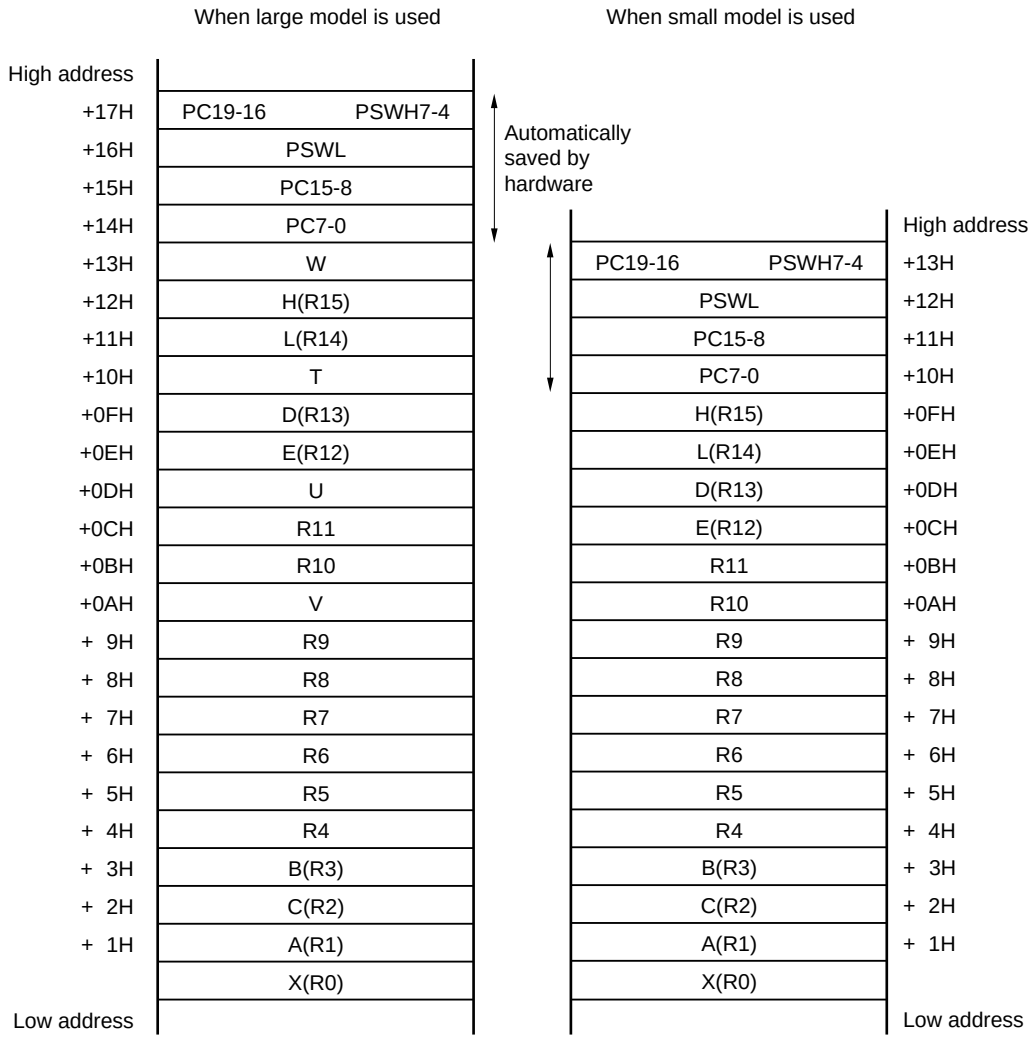
Caution When this system call is issued from an interrupt handler, be sure to save the register values before the interrupt occured to the stack. Save the register values to the stack in the sequence shown in Fig. 4-2. If the interrupt handler occupies a register bank, select the register bank of the task executed before the interrupt has occurred, and then call this system call.

[C format]

```
ret_wup (tskid);
char    *tskid;
```

Remark Because this system call is only supported in version 2.00 or above of the C complier CC78K4, it cannot be used with a C complier of a version lower than 2.00. With a C complier of a version 2.00 or above, specify “#pragma∆rtos_interrupt” (refer to **APPENDIX 2**).

Fig. 4-2 Stack Status When ret_int and ret_wup Are Called



4.5 Version Management

The following system call can be used for version management:

(1) get_ver Gets the version number of RX78K/IV

4.5.1 get_ver (Get Version No.)

[Function]

Gets the version number of RX78K/IV.

[Remarks]

Sets the top address of the version management block in an address specified by the parameter. The version management block stores the following information. The area for pk_ver should be allocated by the user. The details of the version information is shown below.

Parameter	Size	Value	Meaning
maker	2 byte	0×000d	Manufacturer (NEC)
id	2 byte	0×0	Model number (Large model)
spver	2 byte	0×5201	TRON specification version (μITRON Ver. 2.01)
prver	2 byte	0×0120	Product version (RX78K/IV Ver. 1.20)
prno	2 byte[*4]	0×0	Product management information
cpu	2 byte	0×0d14	CPU information (NEC 78K/IV series)
var	2 byte	0×0	Variation descriptor

[System call ID number]

get_ver = 25

[Parameter]

pk_ver (Packet of Version Numbers) Address of area storing the pointer to version management block
- 24 bits (large model)
- 16 bits (small model)

[Return parameter]

E_OK Normal termination

[Assembler format]

[Large model]

```
MOV     bnk0_b, #25
MOVW    bnk0_vp, #pk_ver
MOV     bnk0_e, #pk_ver
CALLT   [40H]
        :
```

C register = return parameter

[Small model]

```
MOV     bnk0_b, #25
MOVW    bnk0_vp, #pk_ver
CALLT   [40H]
        :
```

C register = return parameter

[C format]

```
ret = get_ver (pk_ver);
char    **pk_ver;
```

4.6 Time Management

The system calls for time management are as follows:

(1)	act_cyc	Controls activation of a cyclic handler.
	iact_cyc	Controls activation of a cyclic handler. (When used within an interrupt handler)

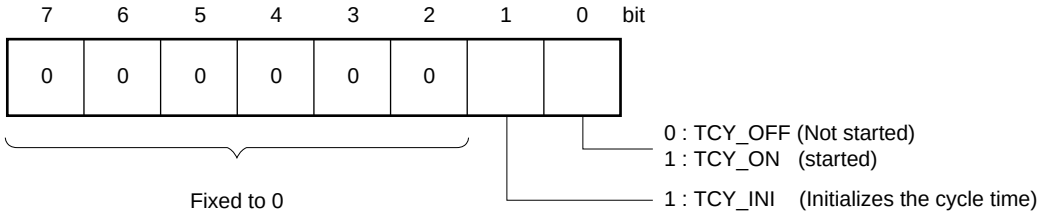
4.6.1 act_cyc (Activate Cyclic Handler)
iact_cyc (Activate Cyclic Handler for Interrupt)

[Function]

Controls activation of a cyclic handler.

[Remarks]

Changes the activation state of the cyclic handler shown by cyhid to the state specified by the cyhact. The specific method of specifying cyhact is as follows:



cyhact = (TCY_OFF || TCY_ON) | [TCY_INI]

“cyhact = TCY_OFF” means that the cyclic handler has temporarily stopped. During this time, the handler is not started.
“cyhact = TCY_ON” sets the activation state to ON, independently of the elapsing of the cycle. The specified cycles are counted even while the activation state is OFF; therefore, the time from execution of the “act_cyc (iact_cyc)” system call until the first execution of the cyclic handler is not constant.
“cyhact = (TCY_ON | TCY_INI)” not only sets the activation state to ON but at the same time clears the cyclic handler counter. Therefore, the initial handler startup takes place exactly after the specified number of cycles has elapsed since the “actcyc (iact_cyc)” system call.

A cyclic handler cannot be created or destroyed dynamically. A cycle handler is registered with the configurator.

[System call ID number]

act_cyc = 26

[Parameter]

cyhid (Cyclic Handler Identifier)	Cyclic handler ID (HCB top address)	- 24 bits (large model) - 16 bits (small model)
cyhact (Cyclic Handler Activation)	Cyclic handler activation state	- 8 bits

[Return parameter]

E_OK	Normal termination

[Assembler format]

- In the case of act_cyc:

[Large model]

```
MOV     bnk0_b, #26
MOV     bnk0_e, #cyhact
MOVW    bnk0_up, #cyhid
MOV     bnk0_d, #tskpri
CALLT   [40H]
        :
```

C register = return parameter

[Small model]

```
MOV     bnk0_b, #26
MOV     bnk0_e, #cyhact
MOVW    bnk0_up, #cyhid
CALLT   [40H]
        :
```

C register = return parameter

- In the case of iact_cyc:

[Large model]

```
MOV     E, #cyhact
MOVG    UUP, #cyhid
CALLT   [4EH]
        :
```

C register = return parameter

[Small model]

```
MOV     E, #cyhact
MOVW    UP, #cyhid
CALLT   [4EH]
        :
```

C register = return parameter

[C format]

- In the case of act_cyc:

```
ret = act_cyc(cyhid, cyhact);
char    cyhid;
char    cyhact;
```

- In the case of iact_cyc:

```
ret = iact_cyc(cyhid, cyhact);
char    cyhid;
char    cyhact;
```

CHAPTER 5 RX78K/IV RESET OPERATION

A system running RX78K/IV is initialized when the system initialization reset routine stored with RX78K/IV in the internal ROM or external memory of the 78K/IV series is expected.

After resetting the device, the system is branched out to the address which is set in address 0000H. Therefore, be sure to set the address of the user's initialization routine. Make sure that the user initial routine performs necessary processing such as initialization of the peripheral hardware. After this, read out the location address of the kernel from the initialization information table created by the configurator, and branch to the reset routine for system initialization. At this time, set the top address of the initialization information table to the TDE register.

Fig. 5-1 shows an example program to branch to the reset routine for system initialization.

The reset routine for system initialization initializes the reserved area and object in accordance with the initialization information table. After initialization, control is passed to the RX78K/IV dispatcher. The dispatcher selects a task from a ready queue initialized by the reset routine and places the task in the RUN state. The task selected at this time is called an initial task.

Fig. 5-1 Reset Routine Branch Program for System Initialization (Large model)

```
          NAME     RES_PROG
;
          EXTRN    SYS_INF,          ; initialization information table
;
VCTTB10   CSEG   AT 0000H
          DW       RES_RTN           ; sets vector address at reset

VCTTBL1   CSEG   AT 0018H
          DW       ?TMDSP            ; sets vector address for timer processing

RESRTN    CSEG
RES_RTN:
          LOCATION 0FH
          (Initialization of RAM, initialization of peripheral hardware, etc.)
          MOVG     TDE, #SYS_INF     ; sets top address of the initialization
                                       information table to TDEreg.
          MOVW     AX, [TDE]         ; sets kernel location address to AXreg.
          BR       AX                ; branches to the reset routine for system
                                       initialization of RX78K/IV
;
          END
```

5.1 Hardware Used

With RX78K/IV, hardware such as the timer/counter unit used for “wai_tsk”, “act_cyc(iact_cyc)” is used (refer to **Table 5-1**).

Since RX78K/IV does not initialize the timer/counter unit, initialize the unit within the user initialization routine.

When “wai_tsk”, “act_cyc(iact_cyc)” is not used, the following timer/counter unit is not used.

Table 5-1 Timer of Each CPU

CPU	Timer	Compare Register	Interrupt Vector
μPD784021, 784025 μPD784026, 78P4026	TM3 (16 bits/timer 3)	CR30	INTC30

To use a timer other than TM3, set the start address (“?tmdsp”) of the timer dispatcher to the vector table address of the timer used (refer to **Fig. 5-1**).

When setting 3 for the interrupt level of the timer processing to permit multiple interrupt with the interrupt handler, this OS cannot warrant proper operations. Therefore, ensure either to set a value other than 3 for the interrupt level of the timer processing or inhibit multiple interrupt if interrupt level 3 is to be set.

APPENDIX 1 ERROR CODE LIST AND TASK STATUS LIST

- Error code list

Error code	Value (8 bits, char type)	Contents
E_OK	0	Normal termination
E_NODMT	1	Specified task is not in DORMANT state when system call “sta_tsk” is issued
E_DMT	2	Specified task is in DORMANT state when system call “wup_tsk(iwup_tsk)”, “ter_tsk”, or “chg_pri” is issued
E_QOVR	3	Queue overflow
E_TMOUT	4	When task is terminated after lapse of specified time when system call “wai_tsk” is issued
E_PLFAIL	5	Polling has failed

- Task status list

Value	Status
0x0	DORMANT status
0x1	RUN or READY status
0x2	WAIT status (timeout)
0x3	WAIT status (sleep)
0x4	WAIT status (eventflag) w/clear
0x5	WAIT status (eventflag) w/o clear
0x6	WAIT status (semaphore)
0x7	WAIT status (message)

[MEMO]

APPENDIX 2 DESCRIPTION IN C LANGUAGE

C Compiler CC78K4 of version 2.00 or above supports tasks and interrupt handlers.
Examples of describing a task and an interrupt handler in the C language are shown below.

- Task

Description format

```
#pragma rtos_task [task function name]
```

Example

```
#pragma rtos_task sample

extern void slp_tsk (void);     /* External reference of slp_tsk */

void sample()
{
       slp_tsk();

       ext_tsk();
}
```

Remark No external reference declaration is necessary for the “ext_tsk” system call.

- Interrupt handler

Description format

```
#pragma rtos_interrupt [(interrupt request name)(interrupt handler name){stack
selection specification]
```

Remark stack selection specification: sp = array name [+offset position]

Example

```
#pragma rtos_interrupt INTPO sample_h sp=int_stack+10

unsigned char int_stack[10];   /* Allocates stack area for interrupt */
extern char iset_flg(char*);   /* External reference declaration of iset_flg */
extern char flag_idl;          /* External reference declaration of event flag ID */

sample_h()
{
      iset_flg(&flag_idl);

      ret_int();
}
```

Remark External reference declaration is not necessary for system calls “ret_int” and “ret_wup”.
For details, refer to the User’s Manual of the C Compiler.

[MEMO]

APPENDIX 3 SYSTEM CALL LIST

System Call	Function	Entry No.
sta_tsk	Starts task by placing it in RUN state from DORMANT state	0
ext_tsk	Normally terminates task itself by placing it in DORMANT state from RUN state	1
ter_tsk	Forcibly terminates another task by placing it DORMANT state	2
chg_pri	Changes a task's priority	3
ichg_pri	Changes a task's priority [Issues from interrupt handler]	–
rot_rdq	Rotates the ready queue of the specified priority	4
irot_rdq	Rotates the ready queue of the specified priority [Issues from interrupt handler]	–
tsk_sts	Checks the task status	5
slp_tsk	Places the task in the WAIT state	6
wai_tsk	Places the task in the WAIT state temporarily	7
wup_tsk	Wakes up and places the task in the READY state from the WAIT state	8
iwup_tsk	Wakes up and places the task in the READY state from the WAIT state [Issues from interrupt handler]	–
can_wup	Cancels a task wake-up request	9
set_flg	Sets an event flag	10
iset_flg	Sets an event flag [Issues from interrupt handler]	–
clr_flg	Clears an event flag	11
wai_flg	Waits for an event flag to be set (w/o clear)	12
cwai_flg	Waits for an event flag to be set (w/clear)	13
pol_flg	Gets an event flag (w/o clear)	14
cpol_flg	Gets an event flag (w/clear)	15
sig_sem	Signal manipulation of a semaphore (V instruction)	16
isig_sem	Signal manipulation of a semaphore (V instruction) [Issues from interrupt handler]	–
wai_sem	Wait manipulation of a semaphore (P instruction)	17
preq_sem	Obtains semaphore resources	18
snd_msg	Sends a message	19
isnd_msg	Sends a message [Issues from interrupt handler]	–
rcv_msg	Waits for reception from a mail box	20
prcv_msg	Receives a message	21
pget_blk	Gets a fixed-length memory block	22
rel_blk	Releases a fixed-length memory block	23
ret_int	Returns from an interrupt handler	–
ret_wup	Returns from an interrupt handler and wakes up a task	–
get_ver	Gets the RX78K/IV version number	25
act_cyc	Controls activation of a cyclic handler.	26
iact_cyc	Controls activation of a cyclic handler. [Issues from interrupt handler]	–

[MEMO]

APPENDIX 4 EXAMPLE OF CYCLIC HANDLER CODING

When creating a cyclic handler, be careful about the following points.

(1) The cyclic handler can be mapped in the 1MB memory space. (Large model only)

(2) The cyclic handler is invoked with the “br rg” command from the timer dispatch routine. For this reason, it is handled in the same manner as an interrupt handler; therefore, the system calls that can be issued are only the “i×××_×××” system calls. However, the “i×××_×××” system calls targeting cyclic handlers cannot be issued from a cyclic handler.

(3) After ending the cyclic handler processing, be sure to return to the timer dispatch routine “?tm_ret”. Please code the cyclic handler processing as follows:

```
        public   [cyclic handler address]
        extern   ?tm_ret

[Cyclic handler address]:
        [Cyclic handler processing]

        br       !?tm_ret
```

(4) When switching to another register bank in the cyclic handler, return to the original register bank first before returning to the timer dispatch routine.

(5) In the timer dispatch routine, the “rg7, rg6, rg5, rp1, and rp0” registers are stored on the stack when interrupted. Therefore, when using the “rg4, rp3, and rp2” registers in the cyclic handler, please restore the value to the original one when returning to the timer dispatch routine.

```
        public  cyc_ptr
        extrn    ?tm_ret

 cyc_ptr:
        push    psw                     ← (4)
        sel     rb6

        push    rp2                     ← (5)

          cyclic handler processing

        pop     rp2                     ← (5)
        pop     psw                     ← (4)
        br      !?tm_ret                ← (3)
```

[MEMO]

APPENDIX 5 ESTIMATING MEMORY CAPACITY (LARGE MODEL)

On RX78K/IV, it is possible to set an area capable of generating objects (TCB, event flag, semaphore, mail box, memory pool, and cyclic handler) in an arbitrary area. However, the area of 20 bytes starting from the start address of the user-set area is an OS management area. Each object management area is secured from there on. Then the 12 bytes to 64 bytes ready queue area is secured. The ready queue area varies depending on the specified number of task priorities. Each object can be generated in up to 255 units.

Calculation of the object size (data) for a system running RX78K/IV is shown below.

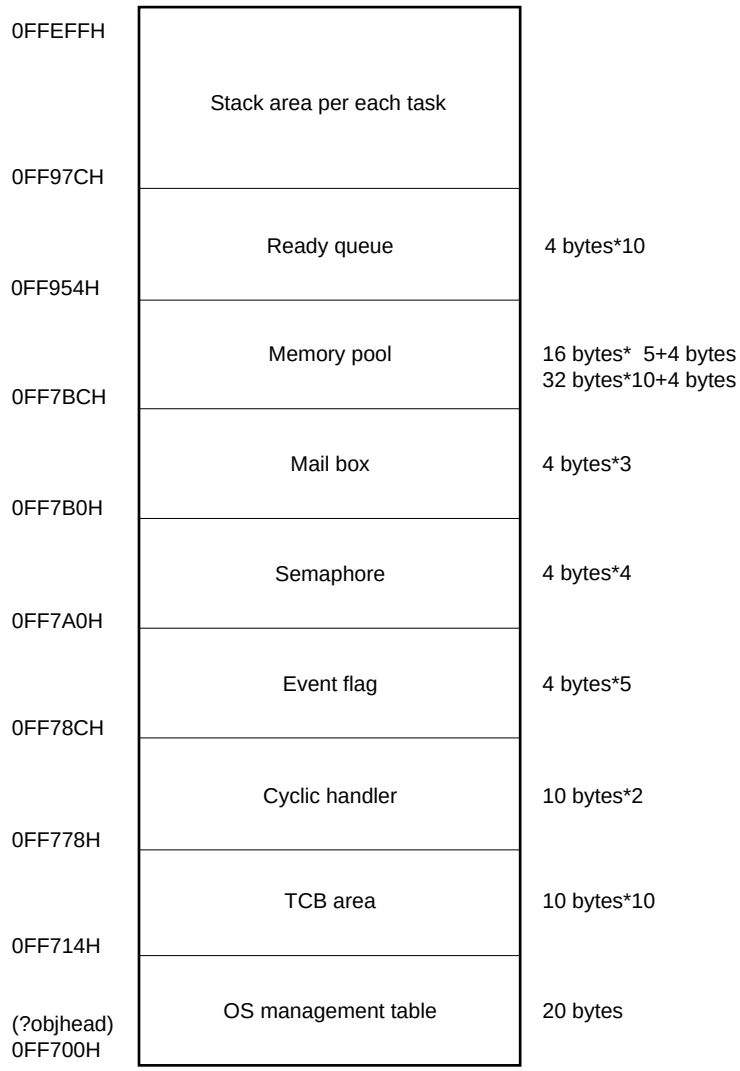
Example Where number of objects to be created is:

Task : 20 Event flag : 5
Semaphore : 4 Mail box : 3
Cyclic handler : 2
Memory pool : 2 (number of memory blocks: 5, size of 1 memory block: 16 bytes)
(number of memory blocks:10, size of 1 memory block: 32 bytes)
The task priority can be used up to 10 levels.
The top address (?objhead) of the OS management area shall be 0FF700H.

RAM size used		OS management table		Ready queue		TCB		Event flag		Semaphore		Mail box		Cyclic handler
↓		↓		↓		↓		↓		↓		↓		↓
228 (bytes)	=	20	+	4 × 10	+	10 × 10	+	4 × 5	+	4 × 4	+	4 × 3	+	10 × 2

RAM size used			Memory pool	
↓			↓	
408 (bytes)	=	4 + 16 × 5	+	4 + 32 × 10

Total 228 + 408 = 636 (bytes)


0FFEFFH
Stack area per each task
0FF97CH
Ready queue
4 bytes*10
0FF954H
Memory pool
16 bytes* 5+4 bytes
32 bytes*10+4 bytes
0FF7BCH
Mail box
4 bytes*3
0FF7B0H
Semaphore
4 bytes*4
0FF7A0H
Event flag
4 bytes*5
0FF78CH
Cyclic handler
10 bytes*2
0FF778H
TCB area
10 bytes*10
0FF714H
OS management table
20 bytes
(?objhead)
0FF700H

APPENDIX 6 ESTIMATING MEMORY CAPACITY (SMALL MODEL)

On RX78K/IV, it is possible to set an area capable of generating objects (TCB, event flag, semaphore, mail box, memory pool, and cyclic handler) in 256 bytes of the internal RAM area 0F700H through 0F7FFH. However, the area of 14 bytes starting from 0F700H is an OS management area. Each object management area is secured from there on. Then the 6 bytes to 32 bytes ready queue area is secured. The ready queue area varies depending on the specified number of task priorities. Each object can be generated in up to 255 units.

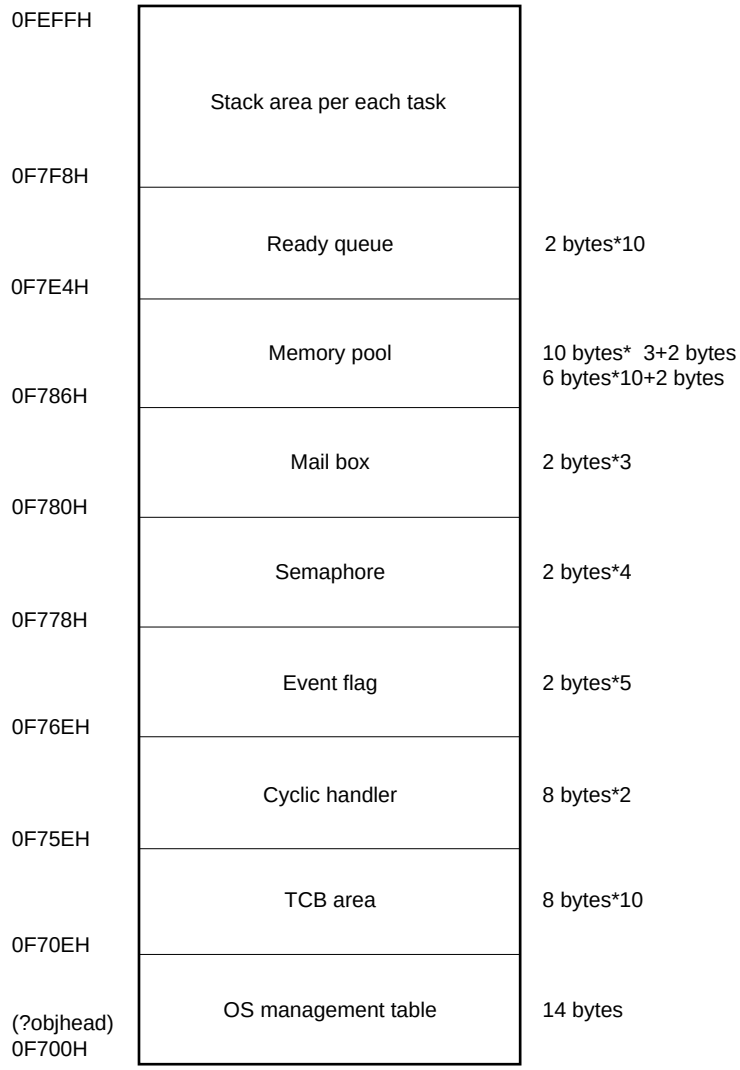
Calculation of the object size (data) for a system running RX78K/IV is shown below.

Example Where number of objects to be created is:

Task : 20 Event flag : 5
Semaphore : 4 Mail box : 3
Cyclic handler : 2
Memory pool : 2 (number of memory blocks: 5, size of 1 memory block: 10 bytes)
(number of memory blocks: 10, size of 1 memory block: 6 bytes)
The task priority can be used up to 10 levels.
The top address (?objhead) of the OS management area shall be 0F700H.

RAM size used		OS management table		Ready queue		TCB		Event flag		Semaphore		Mail box		Cyclic handler
↓		↓		↓		↓		↓		↓		↓		↓
154 (bytes)	=	14	+	2×10	+	8×10	+	2×5	+	2×4	+	2×3	+	8×2

RAM size used			Memory pool	
↓			↓	
94 (bytes)	=	$2 + 10 \times 3$	+	$2 + 6 \times 10$
Total			$154 + 94 = 248$ (bytes)	


0FEFFH
Stack area per each task
0F7F8H
Ready queue
2 bytes*10
0F7E4H
Memory pool
10 bytes* 3+2 bytes
6 bytes*10+2 bytes
0F786H
Mail box
2 bytes*3
0F780H
Semaphore
2 bytes*4
0F778H
Event flag
2 bytes*5
0F76EH
Cyclic handler
8 bytes*2
0F75EH
TCB area
8 bytes*10
0F70EH
OS management table
14 bytes
(?objhead)
0F700H

APPENDIX 7 STANDBY FUNCTION

The RX78K/IV places the CPU in the HALT status as default assumption when there are no longer any selected tasks. In some systems, however, the STOP mode or IDLE mode may be preferred to the HALT mode. With the RX78K/IV, therefore, the standby processing is decided by the user. The supplied sample of the standby block, “nuc_idle.asm”, is shown below.

Sample “nuc_idle.asm”

```
public      @nuc_idl
            cseg        base
@nuc_idl:
            mov         x, stbc
            and         x, #0f0h
            shr         x, 1
            mov         a, #0
            movw        bc, #hlt_tbl
            addw        bc, ax
            br          bc
;
hlt_tbl:
            ei
            mov         stbc, #00000001b            ; set halt mode (fxx/2)
            ret
;
            nop
            nop
;
            ei
            mov         stbc, #00010001b            ; set halt mode (fxx/4)
            ret
;
            nop
            nop
;
            ei
            mov         stbc, #00100001b            ; set halt mode (fxx/8)
            ret
;
            nop
            nop
;
            ei
            mov         stbc, #00110001b            ; set halt mode (fxx/16)
            ret
;
            end
```

Remark The above is the file supplied as a sample. This file is a routine that can be executed by the RX78K/IV even if the internal system clock is changed in the user system.

The following sample indicates an example of changing the internal system clock if the clock is fixed in the user system.

Sample 1

```
public      @nuc_idl
            cseg        base
@nuc_idl:
            ei
            mov         stbc, #00000001b            ; set halt mode (fxx/2)
            ret
;
            end
```

Remark The above sample is for when the internal system clock is fixed. However, do not change symbol name "@nuc_idl".

The default standby mode of the RX78K/IV is HALT. To change the mode, refer to the following:

- To change from HALT mode to STOP mode

```
                    Change standby control register (STBC) value
Before change                       After change
mov      stbc, #00000001b     →     mov     stbc, #00000010b
```

- To change from HALT mode to IDLE mode

```
                    Change standby control register (STBC) value
Before change                       After change
mov      stbc, #00000001b     →     mov     stbc, #00000011b
```

For the standby control register (STBC), refer to the User's Manual of the device.

APPENDIX 8 MAXIMUM STACK SIZE USED BY EACH SYSTEM CALL

The stack area of the task that has issued the system call is used as a stack area used for system call processing. Therefore, the stack size of each system call must be determined taking into consideration the stack size used for the user task and the maximum stack size of the system call.

The stack area which is used for the interrupt handler and the timer processing can be set to an arbitrary address. However, when an interrupt occurs, this task stack area uses 4 bytes.

The maximum stack size for the processing of each system call when the program is described in assembly language is as shown in the following table.

System Call Name	Maximum Stack Size Used (Bytes)			
	Large model		Small model	
	Register Bank Used		Register Bank Used	
	1	2 to 7	1	2 to 7
sta_tsk	28	8	24	8
ext_tsk	28	8	24	8
ter_tsk	28	8	24	8
chg_pri	28	8	24	8
ichg_pri	22	22	19	19
rot_rdq	28	8	24	8
irot_rdq	19	19	14	14
tsk_sts	28	8	24	8
slp_tsk	28	8	24	8
wai_tsk	28	8	24	8
wup_tsk	28	8	24	8
iwup_tsk	18	18	16	16
can_wup	28	8	24	8
set_flg	28	8	24	8
iset_flg	23	23	17	17
clr_flg	28	8	24	8
wai_flg	28	8	24	8
cwai_flg	28	8	24	8
pol_flg	28	8	24	8
cpol_flg	28	8	24	8
sig_sem	28	8	24	8
isig_sem	21	21	16	16
wai_sem	28	8	24	8
preq_sem	28	8	24	8
snd_msg	28	8	24	8
isnd_msg	23	23	20	20
rcv_msg	31	11	26	10
prcv_msg	28	8	24	8
pget_blk	28	8	24	8
rel_blk	28	8	24	8
ret_int	24	24	20	20
ret_wup	24	24	20	20
get_ver	28	8	24	8
act_cyc	28	8	24	8
iact_cyc	10	10	9	9
Timer processing	24	17	20	14

Remark Including 4 bytes (PC, PSW) used at interrupting.

APPENDIX 9 RESTRICTIONS ON SYMBOL NAMES

RX78K/IV uses the following symbols as public symbols. Therefore, make sure that the same names are not used on tasks and interrupt handlers.

?objhead
brproc
?sysrt
?qin
?qin1
?qout
?qout1
?canc1
?cnsav
?itdsp
?tmdsp
?tkdsp
?tm_ret
x_serial
bittbl
ent_tbl
sys_inf
rdyqtp
az_arg
wtcbh
wtcbt
runtcb
timnst
wqlnkp
retdata
pribit
cnftblp
tranos
bittbl
end_intsys
?db_tkds
end_itdsp
?db_itds
@nuc_idl

System call names

[MEMO]