RENESAS

User's Manual

# RI78V4 V2.00.00

## Real-Time Operating System

## User's Manual: Coding

Target Device
RL78 Family

NOTICE :
There are the correction on page 15 and 116 in this document.



Renesas Electronics
www.renesas.com

Rev.1.00 Mar 2015

# How to Use This Manual

**Readers** This manual is intended for users who design and develop application systems using RL78 family microcontrollers products.

**Purpose** This manual is intended for users to understand the functions of real-time OS "RI78V4" manufactured by Renesas Electronics, described the organization listed below.

**Organization** This manual consists of the following major sections.



**How to Read This Manual** It is assumed that the readers of this manual have general knowledge in the fields of electrical engineering, logic circuits, microcontrollers, C language, and assemblers.

To understand the hardware functions of the RL78 family.
-> Refer to the **User's Manual** of each product.

**Conventions**

| | |
|---|---|
| Data significance: | Higher digits on the left and lower digits on the right |
| **Note**: | Footnote for item marked with **Note** in the text |
| **Caution**: | Information requiring particular attention |
| **Remark**: | Supplementary information |
| Numeric representation: | Decimal ... XXXX |
| | Hexadecimal ... 0xXXXX |

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

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

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

**Related Documents**

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

| Document Name | | Document No. |
|---|---|---|
| RI Series | Start | R20UT0751E |
| | Message | R20UT0756E |
| RI78V4 V2.00.00 | Coding | This manual |
| | Debug | R20UT3374E |
| | Analysis | R20UT3373E |

**Caution** **The related documents listed above are subject to change without notice. Be sure to use the latest edition of each document when designing.**

# CHAPTER 1 OVERVIEW

## 1.1 Outline

The RI78V4 is a built-in real-time, multi-task OS that provides a highly efficient real-time, multi-task environment to increases the application range of processor control units.

The RI78V4 is a high-speed, compact OS capable of being stored in and run from the ROM of a target system.

### 1.1.1 Real-time OS

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

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

Real-time OS has been designed to overcome this problem.

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

### 1.1.2 Multi-task OS

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

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

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

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

# TABLE OF CONTENTS

# CHAPTER 2 SYSTEM CONSTRUCTION

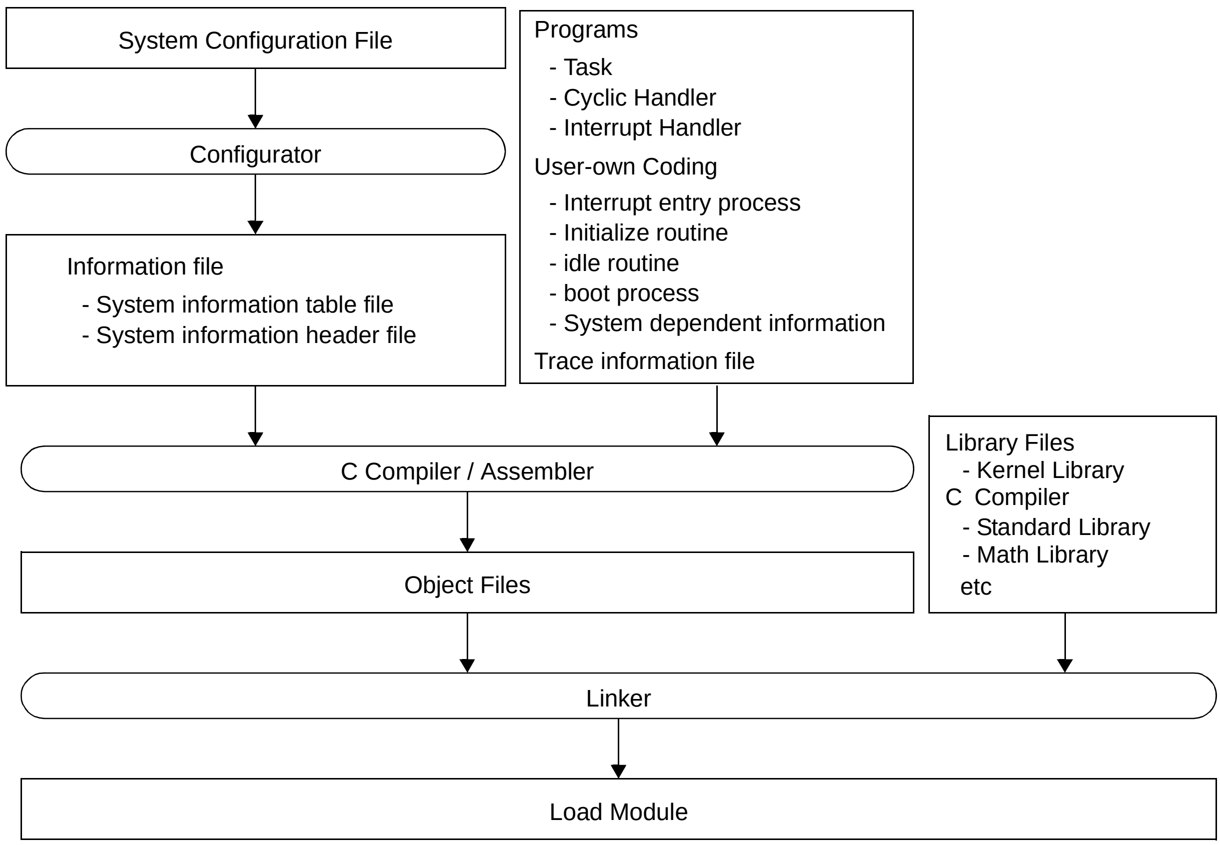
This chapter describes how to build a system (load module) that uses the functions provided by the RI78V4.

## 2.1 Outline

System building consists in the creation of a load module using the files (kernel library, etc.) installed on the user development environment (host machine) from the RI78V4's supply media.

The following shows the procedure for organizing the system.

Figure 2-1 Example of System Construction



## 2.2 Coding of Processing Program

Code the processing that should be implemented in the system.

In the RI78V4, the processing program is classified into the following three types, in accordance with the types and purposes of the processing that should be implemented.

- Tasks
  A task is processing program that is not executed unless it is explicitly manipulated via service calls provided by the RI78V4, unlike other processing programs (cyclic handler and interrupt handler).

  Note For details about the task, refer to "3.2 Tasks".

- Cyclic Handlers
  The cyclic handler is a routine dedicated to cycle processing that is activated periodically at a constant interval (activation cycle).
  The RI78V4 handles the cyclic handler as a "non-task (module independent from tasks)". Therefore, even if a task with the highest priority in the system is being executed, the processing is suspended when a specified activation cycle has come, and the control is passed to the cyclic handler.

  Note For details about the cyclic handler, refer to "7.5 Cyclic Handlers".

- Interrupt Handlers
  The interrupt handler is a routine dedicated to interrupt servicing that is activated when an interrupt occurs.
  The RI78V4 handles the interrupt handler as a "non-task (module independent from tasks)". Therefore, even if a task with the highest priority in the system is being executed, the processing is suspended when an interrupt occurs, and the control is passed to the interrupt handler.

  Note 1 For details about the interrupt handler, refer to "9.3 Interrupt Handlers".

  Note 2 The user must code the interrupt handlers that calls the Timer Handler.

## 2.3 Coding of System Configuration File

Code the SYSTEM CONFIGURATION FILE required for creating information files (system information table file, system information header file, Interrupt information definition file) that contain data to be provided for the RI78V4.

Note For details about the system configuration file, refer to "CHAPTER 13 SYSTEM CONFIGURATION FILE".

## 2.4 Coding of User-Own Coding Module

Code the user-own coding modules that are extracted to allow the RI78V4 to be supported in various execution environments.

In the RI78V4, the user-own coding module is classified into the following four types, in accordance with the types and purposes of the processing that should be implemented.

- Interrupt Entry Processing
  A routine dedicated to entry processing that is extracted from the INTERRUPT MANAGEMENT FUNCTIONS as a user-own coding module to assign instructions to branch to relevant processing (such as Interrupt Handlers or Boot Processing), to the vector table address to which the CPU forcibly passes the control when an interrupt occurs.

  Note 1 For details about the interrupt entry processing, refer to "9.2 Interrupt Entry Processing".

  Note 2 When the interrupt handler is described by C language (the TA_HLNG attribute is specified in a interrupt handler definition of the system configuration file(DEF_INH)), the user does not have to describe interrupt entry processing because of the C compiler outputing "interrupt entry processing which corresponds to an interrupt request name" automatically.

- Boot Processing
  A routine dedicated to initialization processing that is extracted from the SYSTEM CONFIGURATION MANAGEMENT FUNCTIONS as a user-own coding module to initialize the minimum required hardware for the RI78V4 to perform processing. It is called from Interrupt Entry Processing that is assigned to the vector table address to which the CPU forcibly passes the control when a reset interrupt occurs.

  Note For details about the boot processing, refer to "10.2 Boot Processing".

- Initialization Routine
  A routine dedicated to initialization processing that is extracted from the SYSTEM CONFIGURATION MANAGEMENT FUNCTIONS as a user-own coding module to initialize the hardware dependent on the user execution environment (such as the peripheral controller), and is called from the Kernel Initialization Module.

  Note For details about the initialization routine, refer to "10.3 Initialization Routine".

- Idle Routine
  A routine dedicated to idle processing that is extracted from the SCHEDULER as a user-own coding module to utilize the standby function provided by the CPU (to achieve the low-power consumption system), and is called from the scheduler when there no longer remains a task subject to scheduling by the RI78V4 (task in the RUNNING or READY state) in the system.

  Note For details about the idle routine, refer to "11.7 Idle Routine".

## 2.5 Start address of section

Specifies the start address of section by the user to fix the address allocation done by the linker. In the RI78V4, the allocation destinations (section names) of management objects modularized for each function are specified.

The following lists the section names prescribed in the RI78V4.

Table 2-1 RI78V4 Section

| Section Name | ROM/RAM | Relocation Attribute | Description |
|---|---|---|---|
| .kernel_system | Code flash area | TEXTF | Area where the RI78V4's core processing part and main processing part of service calls provided by the RI78V4 are to be allocated.<br>The start can be aligned at an even address in the area from 0x000c0 to 0xeffff. |
| .kernel_system_timer_n | Code flash area | TEXT | Area where the interrupt for system timer and information of FAR branch are to be allocated.<br>The start can be aligned at an even address in the area from 0x000c0 to 0x0ffff. |
| .kernel_info | Code flash area | CONSTF | Area where information items such as the RI78V4 version are to be allocated.<br>The start can be aligned at an even address that does not span a 64K-1 boundary. |
| .kernel_const<br>.kernel_const_f | Code flash area | CONSTF | Area where initial information items related to OS resources that do not change dynamically are allocated as system information tables and Interrupt infomation definition file.<br>The start can be aligned at an even address that does not span a 64K-1 boundary.<br>This section should be located at 64k boundary + 4 or later. |
| .kernel_stack | RAM area | BSS | Area where the system stack and the task stack are to be allocated.<br>The start can be aligned at an even address in the built-in RAM area from 0xf0000 to 0xfffff and that does not span a 64K-1 boundary. |
| .kernel_data | RAM area | BSS | Area where information items required to implement the functionalities provided by the RI78V4 and information items related to OS resources that change dynamically are allocated as management objects.<br>The start can be aligned at an even address in the built-in RAM area from 0xf0000 to 0xfffff and that does not span a 64K-1 boundary. |
| .kernel_data_init | RAM area | BSS | Area where initial information items of RI78V4 are to be allocated.<br>The start can be aligned at an even address in the built-in RAM area from 0xf0000 to 0xfffff and that does not span a 64K-1 boundary. |
| .kernel_work0<br>.kernel_work1<br>.kernel_work2<br>.kernel_work3 | RAM area | BSS | Area where data queues and fixed-sized memory pools are to be allocated.<br>The start can be aligned at an even address in the built-in RAM area from 0xf0000 to 0xfffff and that does not span a 64K-1 boundary. |
| .kernel_data_trace_n | RAM area | BSS | Area where the trace data are to be allocated.<br>The start can be aligned at an even address in the built-in RAM area from 0xf0000 to 0xfffff and that does not span a 64K-1 boundary. |

| Section Name | ROM/RAM | Relocation Attribute | Description |
|---|---|---|---|
| .kernel_const_trace_f | Code flash area | CONSTF | Area where information items to get trace data are to be allocated. The start can be aligned at an even address that does not span a 64K-1 boundary. |
| .kernel_system_trace_f | Code flash area | TEXTF | Area where the processing part to get trace data are to be allocated. The start can be aligned at an even address in the area from 0x000c0 to 0xeffff. |
| .kernel_sbss | RAM area | SBSS | SADDR area where the RI78V4's core processing are to be allocated. The start can be aligned at an even address in the saddr area. |

Note 1 Specification of .kernel_work0, .kernel_work1, .kernel_work2 and .kernel_work3 is required only when the relevant section names are specified in Data queue information and Fixed-sized memory pool information.

Note 2 The RI78V4 occupies the 8-byte area from the saddr area (0xffe20 to 0xfff1f). Therefore, the available saddr area for the user is up to 247 bytes.

Note 3 The section for RI78V4 is set automatically on CS+. When you want to change the start address of section, changes in the linker setting. For details about the directive file, refer to “CS+ Integrated Development Environment User's Manual: RL78 Building”.

Note 4 For details about the directive file, refer to “CS+ Integrated Development Environment User's Manual: RL78 Coding”.

### 2.5.1 .kernel_system section

The size of the .kernel_system section is approximately 1 KB to 9 KB depends on the service calls used in the processing program.

### 2.5.2 .kernel_system_timer_n section

The following shows an expression required for estimating the .kernel_system_timer_n section size (unit: bytes).

system_timer_n = 16 + (*inthnum_FAR* * 8)

*inthnum_FAR*: Total amount of Interrupt handler information with TA_FAR attribute

### 2.5.3 .kernel_system_trace_f section

The following shows an expression required for estimating the .kernel_system_trace_f section size (unit: bytes).

[ When the trace mode is “Not tracing” ]

system_trace_f = 0

[ When the trace mode is “Takes in trace chart by hardware trace mode” ]

system_trace_f = 184

[ When the trace mode is “Takes in trace chart by software trace mode” ]

system_trace_f = 706

[ When the trace mode is “Takes in long-statistics by software trace mode” ]

system_trace_f = 590

### 2.5.4 .kernel_info section

The size of the .kernel_info section is approximately 16 bytes.

### 2.5.5 .kernel_const section

The following shows an expression required for estimating the .kernel_const section size (unit: bytes).

const = ( *tsknum* * 10 ) + *semnum* + *flgnum* + ( *dtqnum* * 5 ) + ( *mpfnum* * 8 ) + ( *cycnum* * 12 ) + ( *kindnum* * 4 ) + 16

*tsknum*: Total amount of Task information
*semnum*: Total amount of Semaphore information
*flgnum*: Total amount of Eventflag information
*dtqnum*: Total amount of Data queue information
*mpfnum*: Total amount of Fixed-sized memory pool information
*kindnum*: Total number of types defined in the system configuration file among five types of information related to OS resources (Semaphore information, Eventflag information, Data queue information, Mailbox information, Fixed-sized memory pool information and Cyclic handler information)

### 2.5.6 .kernel_const_f section

The following shows an expression required for estimating the .kernel_const_f section size (unit: bytes).

[ When the trace mode is “Not tracing” ]

const_f = 0

[ When the trace mode is “Takes in trace chart by hardware trace mode” ]

const_f = 0

[ When the trace mode is “Takes in trace chart by software trace mode” ]

const_f = 0

[ When the trace mode is “Takes in long-statistics by software trace mode” ]

const_f = 63

### 2.5.7 .kernel_stack section

The following shows an expression required for estimating the .kernel_stack section size (unit: bytes).

$$stack = \sum_{k=1}^{tsknum} ( stksz_k + 20 ) + ( sys\_stksz + 2 )$$

*tsknum*: Total amount of Task information
$stksz_k$: Stack size specified in Task information
*sys_stksz*: Stack size specified in System stack information. When multiple interrupt occurs, adds 18 bytes every one time.

### 2.5.8 .kernel_data section

The following shows an expression required for estimating the .kernel_data section size (unit: bytes).
The expression varies depending on whether or not Semaphore information is defined in the system configuration file.

[ When semaphore information is defined ]

data = align2 ( *maxtpri* + 1 ) + align2 { ( *tsknum* * 24 ) + ( *semnum* * 2 ) + 1 } + align2 ( *flgnum* * 3 ) + align2 { ( *dtqnum* * 4 ) + 1} + ( *mbxnum* * 8 ) + align2 ( *primbx* ) + ( *mpfnum* * 4 ) + ( *cycnum* * 8 ) + 20

[ When semaphore information is not defined ]

data = align2 ( *maxtpri* + 1 ) + ( *tsknum* * 24 ) + align2 ( *flgnum* * 3 ) + align2 { ( *dtqnum* * 4 ) + 1} + ( *mbxnum* * 8 ) + align2 ( *primbx* ) + ( *mpfnum* * 4 ) + ( *cycnum* * 8 ) + 20

| | |
|---|---|
| *maxtpri*: | Priority range specified in Task priority information |
| *tsknum*: | Total amount of Task information |
| *semnum*: | Total amount of Semaphore information |
| *flgnum*: | Total amount of Eventflag information |
| *dtqnum*: | Total amount of Data queue information |
| *mbxnum*: | Total amount of Mailbox information |
| *primbx*: | Total amount of Mailbox information for which the priority is specified for the attribute (message queuing method) |
| *mpfnum*: | Total amount of Fixed-sized memory pool information |
| *cycnum*: | Total amount of Cyclic handler information |

### 2.5.9 .kernel_data_init section

The size of the .kernel_data_init section is approximately 2 bytes.

### 2.5.10 .kernel_work0, .kernel_work1, .kernel_work2, .kernel_work3 section

The following shows an expression required for estimating the size of the .kernel_work0, .kernel_work1, .kernel_work2, and .kernel_work3 section (unit: bytes).

$$\mathrm{work}X = \sum_{k=1}^{mpfnum} ( blkcnt_k \times blksz_k ) + \sum_{k=1}^{dtqnum} ( dtqcnt_k \times 4 )$$

| | |
|---|---|
| *mpfnum*: | Total number of section units for Fixed-sized memory pool information |
| $blkcnt_k$: | Number of fixed-sized memory blocks specified in Fixed-sized memory pool information |
| $blksz_k$: | Block size specified in Fixed-sized memory pool information |
| *dtqnum*: | Total number of section units for Data queue information |
| $dtqcnt_k$: | Number of datq queue specified in Data queue information |

### 2.5.11 .kernel_data_trace_n section

The following shows an expression required for estimating the .kernel_data_trace_n section size (unit: bytes).

[ When the trace mode is “Not tracing” ]

data_trace_n = 0

[ When the trace mode is “Takes in trace chart by hardware trace mode” ]

data_trace_n = 2

[ When the trace mode is “Takes in trace chart by software trace mode” ] data_trace_n = 8 + *bufsize*

*bufsize*: Trace buffer size

[ When the trace mode is “Takes in long-statistics by software trace mode” ]

data_trace_n = { ( *tsknum* + 1) * 20 ) } + { ( *inhnum* + 1)* 8 ) } + 34

*tsknum*: Total amount of Task information
*inhnum*: Total amount of Interrupt handler information

### 2.5.12 .kernel_const_trace_f section

The following shows an expression required for estimating the .kernel_const_trace_f section size (unit: bytes).

[ When the trace mode is “Not tracing” ]

const_trace_n = 6

[ When the trace mode is “Takes in trace chart by hardware trace mode” ]

const_trace_n = 64

[ When the trace mode is “Takes in trace chart by software trace mode” ]

const_trace_n = 70

[ When the trace mode is “Takes in long-statistics by software trace mode” ]

const_trace_n = 70

## 2.6 Creating Load Module

Run a build on the CS+ for files created in sections from "2.2 Coding of Processing Program" to "2.5 Start address of section", and library files provided by the RI78V4 and C compiler package, to create a load module.
The following lists the files required for creating load modules.

1 ) Create or load a project

Create a new project, or load an existing one.

Note See “RI Series Real-Time Operating System User's Manual: Start” or “CS+ Integrated Development Environment User's Manual: Start” for details about creating a new project or loading an existing one.

2 ) Set a build target project

When making settings for or running a build, set the active project.
If there is no subproject, the project is always active.

Note See “CS+ Integrated Development Environment User's Manual: Build” for details about setting the active project.

3 ) Set build target files

For the project, add or remove build target files and update the dependencies.

Note See “CS+ Integrated Development Environment User's Manual: Build” for details about adding or removing build target files for the project and updating the dependencies.

The following lists the files required for creating a load module.

- C/assembly language source files created in "2.2 Coding of Processing Program"
  - Tasks, Cyclic Handlers, Interrupt Handlers
- System configuration file created in “2.3 Coding of System Configuration File”
  - SYSTEM CONFIGURATION FILE

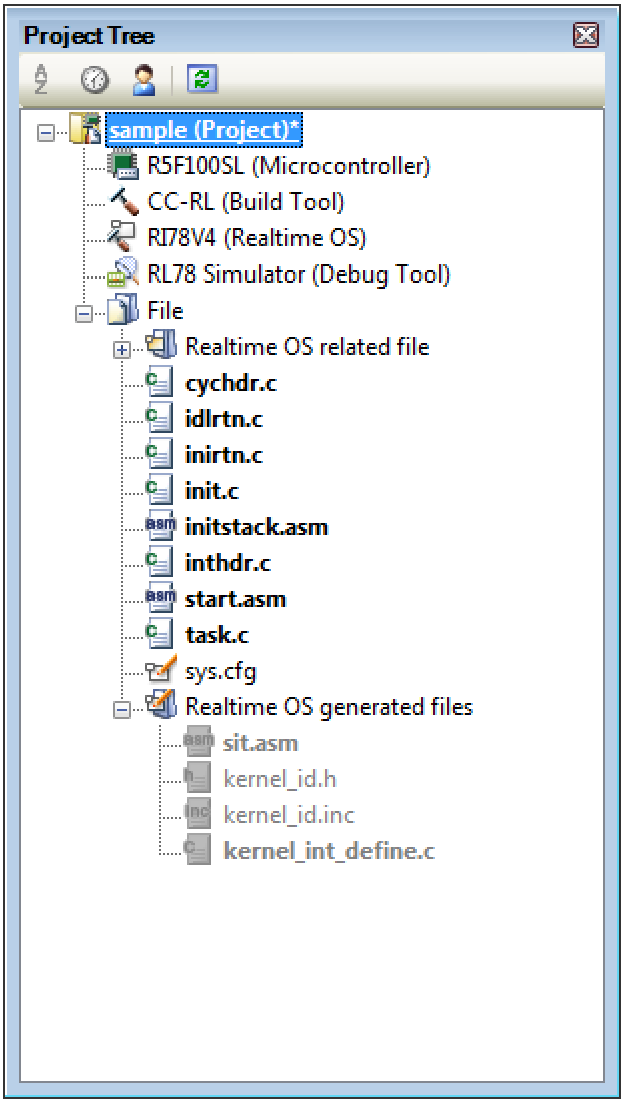
  Note Specify “cfg“ as the extention of the system configuration file name.
  If the extension is different, "cfg" is automatically added (for example, if you designate "aaa.c" as a file name, the file is named as "aaa.c.cfg").
- C/assembly language source files created in "2.4 Coding of User-Own Coding Module"
  - Interrupt Entry Processing, Boot Processing, Initialization Routine, Idle Routine
- Directive file created in "2.5 Start address of section"
  - Directive file
- Files provided by the RI78V4
  - Trace information file
- Library files provided by the RI78V4
  - Kernel library
- Library files provided by the C compiler/assembler package
  - Standard library, runtime library, etc.

Note 1 If the system configuration file is added to the Project Tree panel, the Realtime OS generated files node is appeared.

The following information files are appeared under the Realtime OS generated files node. However, these files are not generated at this point in time.

- System information table file
- System information header file (for C language)
- System information header file (for assembly language)
- Interrupt Information definition file

Figure 2-2 Project Tree Panel (After Adding sys.cfg)



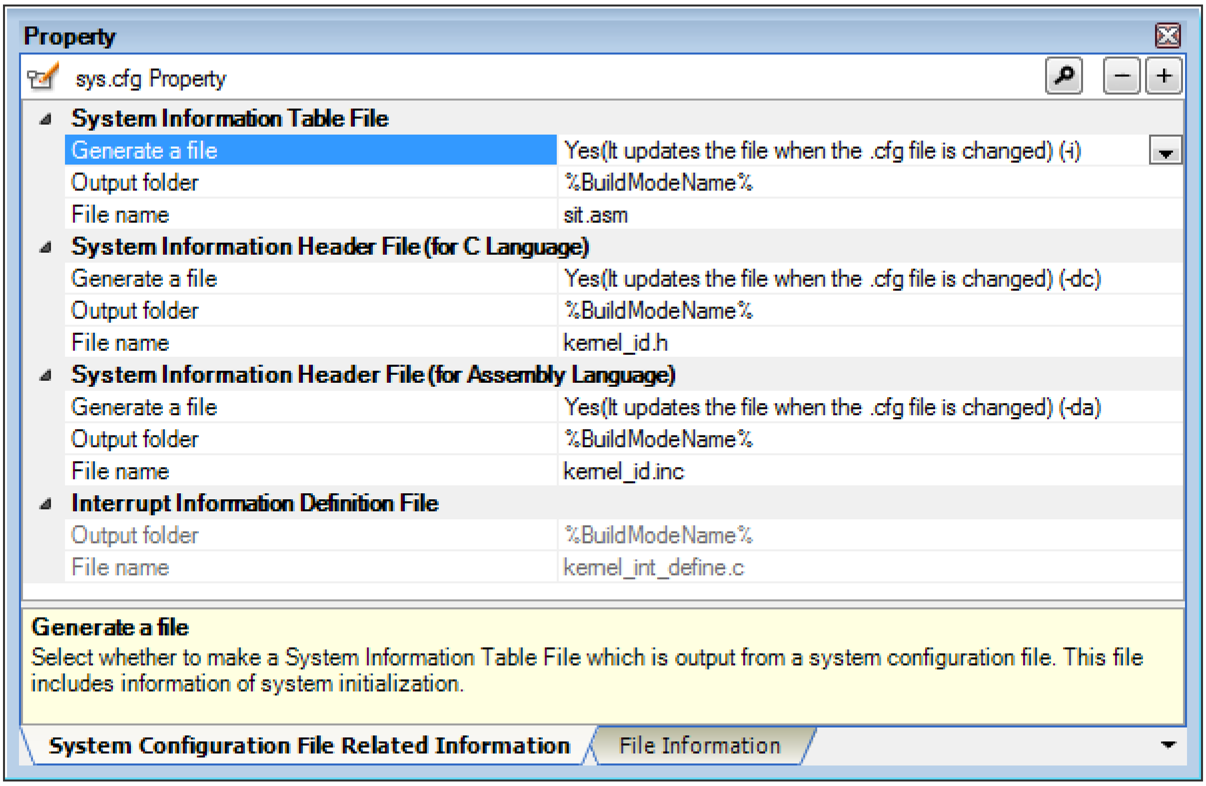

Note 2 When replacing the system configuration file, first remove the added system configuration file from the project, then add another one again.

Note 3 Although it is possible to add more than one system configuration files to a project, only the first file added is enabled. Note that if you remove the enabled file from the project, the remaining additional files will not be enabled; you must therefore add them again.

4 ) Set the output of information files

Select the system configuration file on the project tree to open the Property panel.
On the [System Configuration File Related Information] tab, set the output of information files (system information table file and system information header files).

Figure 2-3 Property Panel: [System Configuration File Related Information] Tab




5 ) Specify the output of a load module file

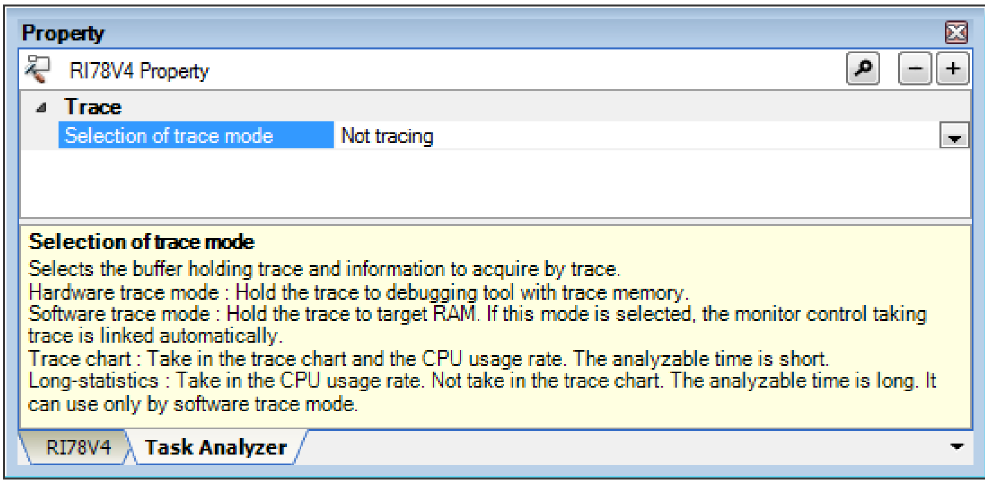
Set the output of a load module file as the product of the build.

Note See “CS+ Integrated Development Environment User's Manual: RL78 Build” for details about specifying the output of a load module file.

6 ) Set trace information

Set the detailed information on the using task analyzer in RI78V4 package.

Figure 2-4 [Task Analyzer] Tab




7 ) Set build options

Set the options for the compiler, assembler, linker, and the like.

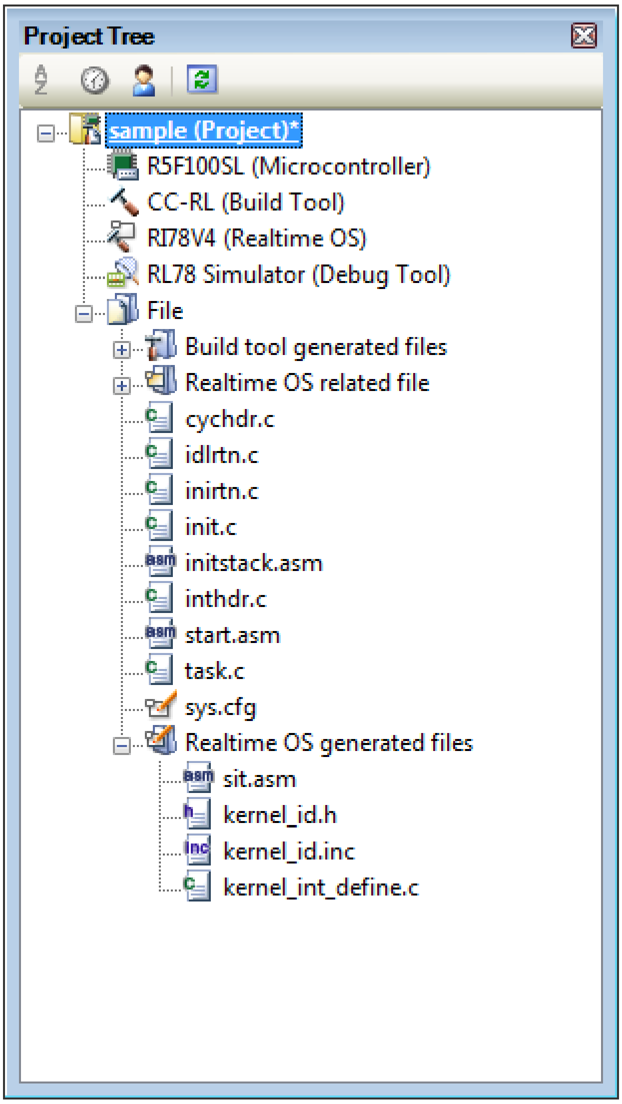
Note See “CS+ Integrated Development Environment User’s Manual: RL78 Build” for details about setting build options.

8 ) Run a build

Run a build to create a load module.

Note See “CS+ Integrated Development Environment User's Manual: RL78 Build” for details about runnig a build.

Figure 2-5 Project Tree Panel (After Running Build)




9 ) Save the project

Save the setting information of the project to the project file.

Note See “CS+ Integrated Development Environment User's Manual: Start” for details about saving the project.

## 2.7 Embedding System

If the output of hex files are set in 4 ) of "2.6 Creating Load Module", hex files are created.
After that, embed the modules to the system by using a flash programmer.

# CHAPTER 3 TASK MANAGEMENT FUNCTIONS

This chapter describes the task management functions performed by the RI78V4.

## 3.1 Outline

The task control functions provided by the RI78V4 include a function to reference task statuses, in addition to a function to manipulate task statuses.

## 3.2 Tasks

A task is processing program that is not executed unless it is explicitly manipulated via service calls provided by the RI78V4, unlike other processing programs (cyclic handler and interrupt handler), and is called from the scheduler.

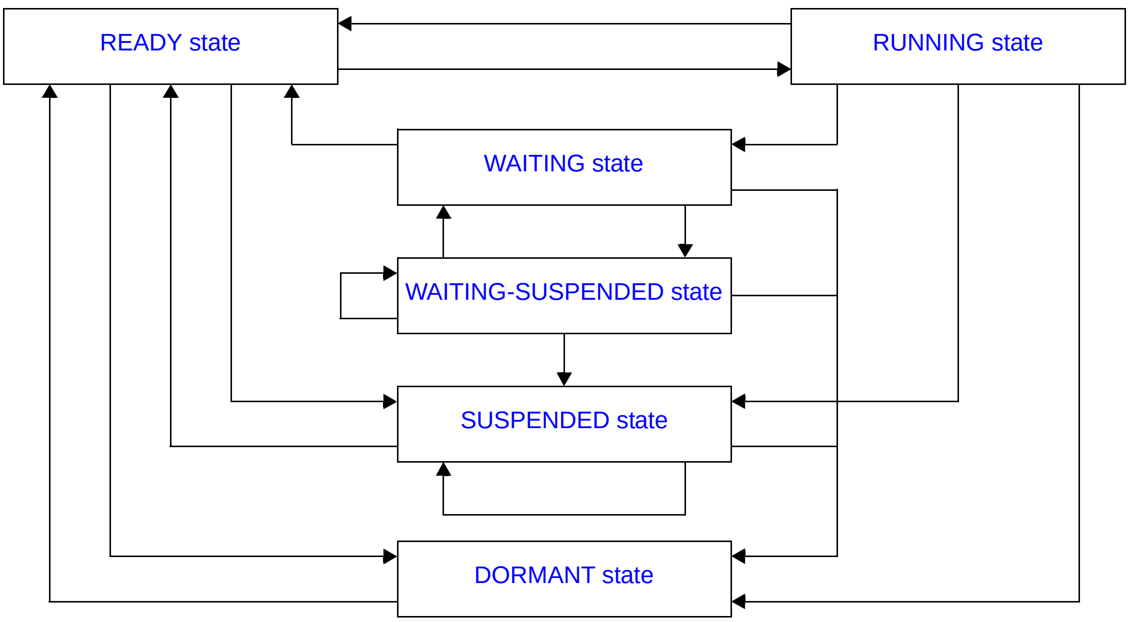
Note The execution environment information required for a task's execution is called "task context". During task execution switching, the task context of the task currently under execution by the RI78V4 is saved and the task context of the next task to be executed is loaded.

### 3.2.1 Task state

Tasks enter various states according to the acquisition status for the OS resources required for task execution and the occurrence/non-occurrence of various events. In this process, the current state of each task must be checked and managed by the RI78V4.

The RI78V4 classifies task states into the following six types.

Figure 3-1 Task State



- DORMANT state

  State of a task that is not active, or the state entered by a task whose processing has ended.
  A task in the DORMANT state, while being under management of the RI78V4, is not subject to the RI78V4 scheduling.

- READY state

  State of a task for which the preparations required for processing execution have been completed, but since another task with a higher priority level or a task with the same priority level is currently being processed, the task is waiting to be given the CPU's use right.

- RUNNING state

  State of a task that has acquired the CPU use right and is currently being processed.
  Only one task can be in the running state at one time in the entire system.

- WAITING state

  State in which processing execution has been suspended because conditions required for execution are not satisfied.
  Resumption of processing from the WAITING state starts from the point where the processing execution was suspended. The value of information required for resumption (such as task context) immediately before suspension is therefore restored.
  In the RI78V4, the WAITING state is classified into the following six types according to their required conditions and managed.

Table 3-1 Waiting States

| Waiting States | Description |
|---|---|
| Sleeping state | A task enters this state if the counter for the task (registering the number of times the wakeup request has been issued) indicates 0x0 upon the issuance of a slp_tsk or tslp_tsk. |
| Delayed state | A task enters this state upon the issuance of a dly_tsk. |
| Waiting state for a semaphore resource | A task enters this state if it cannot acquire a resource from the relevant semaphore upon the issuance of a wai_sem or twai_sem. |
| Waiting state for an eventflag | A task enters this state if a relevant eventflag does not satisfy a predetermined condition upon the issuance of a wai_flg or twai_flg. |
| Sending WAITING state for a data queue | A task enters this state if cannot send a data to the relevant data queue upon the issuance of a snd_dtq or .tsnd_dtq |
| Receiving WAITING state for a data queue | A task enters this state if cannot receive a data from the relevant data queue upon the issuance of a rcv_dtq or trcv_dtq. |
| Receiving waiting state for a mailbox | A task enters this state if cannot receive a message from the relevant mailbox upon the issuance of a rcv_mbx or trcv_mbx. |
| Waiting state for a fixed-sized memory block | A task enters this state if it cannot acquire a fxed-sized memory block from the relevant memory pool upon the issuance of a get_mpf or tget_mpf. |

- SUSPENDED state

  State in which processing execution has been suspended forcibly.
  Resumption of processing from the SUSPENDED state starts from the point where the processing execution was suspended. The value of information required for resumption (such as task context) immediately before suspension is therefore restored.

- WAITING-SUSPENDED state

  State in which the WAITING and SUSPENDED states are combined.

A task enters the SUSPENDED state when the WAITING state is cancelled, or enters the WAITING state when the SUSPENDED state is cancelled.

### 3.2.2 Task priority

A priority level that determines the order in which that task will be processed in relation to the other tasks is assigned to each task.

As a result, in the RI78V4, the task that has the highest priority level of all the tasks that have entered an executable state (RUNNING state or READY state) is selected and given the CPU use right.

In the RI78V4, the following two types of priorities are used for management purposes.

- Task initial priority
  Priority set when a task is created.

- Task current priority
  This is the general term used to describe the priority level of a task from the time it enters the READY state from the DORMANT state until it returns to the DORMANT state.
  Therefore, the current priority level of a task that enters the READY state from the DORMANT state has the same value as the "initial priority level," and the current priority level when the priority level is changed by issuing chg_pri or ichg_pri is the same value as the "priority level after change".

Note 1 In the RI78V4, a task having a smaller priority number is given a higher priority.

Note 2 The priority that can be specified in a system is in the priority range specified in Task priority information.

### 3.2.3 Create task

In the RI78V4, the method of creating a task is limited to "static creation by the Kernel Initialization Module".

Tasks therefore cannot be created dynamically using a method such as issuing a service call from a processing program.

- Static create
  Static task creation is realized by defining Task information in the system configuration file.
  The RI78V4 executes task creation processing based on data stored in information files, using the Kernel Initialization Module, and handles the created tasks as management targets.

### 3.2.4 Delete task

In the RI78V4, tasks created statically by the Kernel Initialization Module cannot be deleted dynamically using a method such as issuing a service call from a processing program.

### 3.2.5 Basic form of tasks

When coding a task, use a void function with one VP_INT argument (any function name is fine) .

The extended information specified with Task information, or the start code specified when sta_tsk or ista_tsk is issued, is set for the *exinf* argument.

The following shows the basic form of tasks.

[ C Language ]

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    /* ............ */          /*Main processing*/

    ext_tsk ( );                /*Terminate invoking task*/
}
```

Note The the #pragma rtos_task directive is defined in the file “kernel_id.h” (CF78V4 outputs automatically). Therefore please the file “kernel_id.h” be sure to do include.

Assembly Language ]

```
$INCLUDE    (kernel.inc)        ;Standard header file definition
$INCLUDE    (kernel_id.inc)     ;System information header file definition

    .PUBLIC  _func_task
    .SECTION .text, TEXT
_func_task:
    PUSH    BC          ;Stores the higher 2 bytes of argument exinf into stack
    PUSH    AX          ;Stores the lower 2 bytes of argument exinf into stack

    ; ............              ;Main processing

    BR      !!_ext_tsk          ;Terminate invoking task
```

### 3.2.6 Internal processing of task

In the RI78V4, original dispatch processing (task scheduling) is executed during task switching.
Therefore, note the following points when coding tasks.

- Coding method
  Code tasks using C or assembly language in the format shown in "3.2.5 Basic form of tasks".

- Stack switching
  In the RI78V4, switching to the stack for the switching destination task (task stack) is executed during task switching. The user is therefore not required to code processing related to stack switching in tasks.

- Interrupt status
  In the RI78V4, the initial interrupt state specified in Task information when a task is switched from the READY state to the RUNNING state.
  To change (disable or enable) the interrupt status in the task, calling of the __DI or __EI function are therefore required.

- Service call issuance
  Service calls that can be issued in tasks are limited to the service calls that can be issued from tasks.

  Note For details on the valid issuance range of each service call, refer to Table 12-8 to Table 12-17.

## 3.3 Activate Task

The RI78V4 provides two types of interfaces for task activation: queuing an activation request queuing and not queuing an activation request.

### 3.3.1 Queuing an activation request

A task (queuing an activation request) is activated by issuing the following service call from the processing program.

- act_tsk, iact_tsk
  These service calls move a task specified by parameter *tskid* from the DORMANT state to the READY state.
  As a result, the target task is queued at the end on the ready queue corresponding to the initial priority and becomes subject to scheduling by the RI78V4.
  If the target task has been moved to a state other than the DORMANT state when this service call is issued, this service call does not move the state but increments the activation request counter (by added 0x1 to the wakeup request counter).
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ID      tskid = ID_tskA;    /*Declares and initializes variable*/

    /* ............ */

    act_tsk ( tskid );          /*Activate task (queues an activation request)*/

    /* ............ */
}
```

Note 1 The activation request counter managed by the RI78V4 is configured in 7-bit widths. If the number of activation requests exceeds the maximum count value 127 as a result of issuing this service call, the counter manipulation processing is therefore not performed but "E_QOVR" is returned.

Note 2 An extended information "Extended information: exinf" is passed to the task activated by issuing this service call.

### 3.3.2 Not queuing an activation request

A task (not queuing an activation request) is activated by issuing the following service call from the processing program.

- sta_tsk, ista_tsk
  These service calls move a task specified by parameter *tskid* from the DORMANT state to the READY state.
  As a result, the target task is queued at the end on the ready queue corresponding to the initial priority and becomes subject to scheduling by the RI78V4.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ID      tskid = ID_tskA;    /*Declares and initializes variable*/
    VP_INT  stacd = 1048575;    /*Declares and initializes variable*/

    /* ............ */

    sta_tsk ( tskid, stacd );   /*Activate task (does not queue an activation
                                  request)*/

    /* ............ */
}
```

Note 1 This service call does not perform queuing of activation requests. If the target task is in a state other than the DORMANT state, the counter manipulation processing is therefore not performed but "E_OBJ" is returned.

Note 2 An start code "*stacd*" is passed to the task activated by issuing this service call.

## 3.4 Cancel Task Activation Requests

An activation request is cancelled by issuing the following service call from the processing program.

- can_act
  This service call cancels all of the activation requests queued to the task specified by parameter *tskid* (sets the activation request counter to 0x0).
  When this service call is terminated normally, the number of cancelled activation requests is returned.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ER_UINT ercd;               /*Declares variable*/
    ID      tskid = ID_tskA;    /*Declares and initializes variable*/

    /* ............ */

    ercd = can_act ( tskid );   /*Cancel task activation requests*/

    if ( ercd >= 0x0 ) {
        /* ............ */      /*Normal termination processing*/
    }

    /* ............ */
}
```

## 3.5 Terminate Task

The RI78V4 provides two types of interfaces for task termination: termination of invoking task and forced termination of other tasks.

### 3.5.1 Terminate invoking task

An invoking task is terminated by issuing the following service call from the processing program.

- ext_tsk
  This service call moves an invoking task from the RUNNING state to the DORMANT state.
  As a result, the invoking task is unlinked from the ready queue and excluded from the RI78V4 scheduling subject.
  If an activation request has been queued to the invoking task (the activation request counter is not set to 0x0) when this service call is issued, this service call moves the task from the RUNNING state to the DORMANT state, decrements the wakeup request counter (by subtracting 0x1 from the wakeup request counter), and then moves the task from the DORMANT state to the READY state.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    /* ............ */

    ext_tsk ( );                /*Terminate invoking task*/
}
```

Note 1 This service call does not return the OS resource that the invoking task acquired by issuing a service call such as sig_sem or get_mpf. The OS resource have been acquired must therefore be returned before issuing this service call.

Note 2 When moving a task from the RUNNING state to the DORMANT state, this service call initializes the following information to values that are set during task creation.

- Priority (current priority)
- Wakeup request count
- Suspension count
- Interrupt status

Note 3 If the return instruction is written in a task, it executes the same operation as this service call.

Note 4 In the RI78V4, code efficiency is enhanced by coding the return instruction as a "Terminate invoking task".

### 3.5.2 Terminate task

Other tasks are forcibly terminated by issuing the following service call from the processing program.

- ter_tsk
  This service call forcibly moves a task specified by parameter *tskid* to the DORMANT state.
  As a result, the target task is excluded from the RI78V4 scheduling subject.
  If an activation request has been queued to the target task (the activation request counter is not set to 0x0) when this service call is issued, this service call moves the task to the DORMANT state, decrements the wakeup request counter (by subtracting 0x1 from the wakeup request counter), and then moves the task from the DORMANT state to the READY state.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ID      tskid = ID_tskA;    /*Declares and initializes variable*/

    /* ............ */

    ter_tsk ( tskid );          /*Terminate task*/

    /* ............ */
}
```

Note 1 This service call does not return the OS resource that the target task acquired by issuing a service call such as sig_sem or get_mpf. The OS resource have been acquired must therefore be returned before issuing this service call.

Note 2 When moving a task to the DORMANT state, this service call initializes the following information to values that are set during task creation.

- Priority (current priority)
- Wakeup request count
- Suspension count
- Interrupt status

## 3.6 Change Task Priority

The priority is changed by issuing the following service call from the processing program.

- chg_pri, ichg_pri
  These service calls change the priority of the task specified by parameter *tskid* (current priority) to a value specified by parameter *tskpri*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ID      tskid = ID_tskA;    /*Declares and initializes variable*/
    PRI     tskpri = 15;        /*Declares and initializes variable*/

    /* ............ */

    chg_pri ( tskid, tskpri );  /*Change task priority*/

    /* ............ */
}
```

Note If the target task is in the RUNNING or READY state after this service call is issued, this service call re-queues the task at the end of the ready queue corresponding to the priority specified by parameter *tskpri*, following priority change processing.

## 3.7 Reference Task State

A task status is referenced by issuing the following service call from the processing program.

- ref_tsk
  Stores task state packet (such as current status) of the task specified by parameter *tskid* in the area specified by parameter *pk_rtsk*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ID      tskid = ID_tskA;    /*Declares and initializes variable*/
    T_RTSK  pk_rtsk;            /*Declares data structure*/
    STAT    tskstat;            /*Declares variable*/
    PRI     tskpri;             /*Declares variable*/
    STAT    tskwait;            /*Declares variable*/
    ID      wobjid;             /*Declares variable*/
    UINT    actcnt;             /*Declares variable*/
    UINT    wupcnt;             /*Declares variable*/
    UINT    suscnt;             /*Declares variable*/

    /* ............ */

    ref_tsk ( tskid, &pk_rtsk );/*Reference task state*/

    tskstat = pk_rtsk.tskstat;  /*Reference task current state*/
    tskpri = pk_rtsk.tskpri;    /*Reference task current priority*/
    tskwait = pk_rtsk.tskwait;  /*Reference reason for waiting*/
    wobjid = pk_rtsk.wobjid;    /*Reference object ID number for which the task is
                                  waiting*/
    actcnt = pk_rtsk.actcnt;    /*Reference activation request count*/
    wupcnt = pk_rtsk.wupcnt;    /*Reference wakeup request count*/
    suscnt = pk_rtsk.suscnt;    /*Reference suspension count*/

    /* ............ */
}
```

Note For details about the task state packet, refer to "12.5.1 Task state packet".

# CHAPTER 4 TASK DEPENDENT SYNCHRONIZATION FUNCTIONS

This chapter describes the task dependent synchronization functions performed by the RI78V4.

## 4.1 Outline

The RI78V4 provides several task-dependent synchronization functions.

## 4.2 Put Task to Sleep

A task is moved to the sleeping state (waiting forever or with timeout) by issuing the following service call from the processing program.

- slp_tsk
  As a result, the invoking task is unlinked from the ready queue and excluded from the RI78V4 scheduling subject.
  If a wakeup request has been queued to the target task (the wakeup request counter is not set to 0x0) when this service call is issued, this service call does not move the state but decrements the wakeup request counter (by subtracting 0x1 from the wakeup request counter).
  The sleeping state is cancelled in the following cases, and then moved to the READY state.

| Sleeping State Cancel Operation | Return Value |
|---|---|
| A wakeup request was issued as a result of issuing wup_tsk. | E_OK |
| A wakeup request was issued as a result of issuing iwup_tsk. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |

The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ER      ercd;               /*Declares variable*/

    /* ............ */

    ercd = slp_tsk ( );         /*Put task to sleep (waiting forever)*/

    if ( ercd == E_OK ) {
        /* ............ */      /*Normal termination processing*/
    } else if ( ercd == E_RLWAI ) {
        /* ............ */      /*Forced termination processing*/
    }

    /* ............ */
}
```

- tslp_tsk

This service call moves an invoking task from the RUNNING state to the WAITING state (sleeping state).
As a result, the invoking task is unlinked from the ready queue and excluded from the RI78V4 scheduling subject.
If a wakeup request has been queued to the target task (the wakeup request counter is not set to 0x0) when this service call is issued, this service call does not move the state but decrements the wakeup request counter (by subtracting 0x1 from the wakeup request counter).
The sleeping state is cancelled in the following cases, and then moved to the READY state.

| Sleeping State Cancel Operation | Return Value |
|---|---|
| A wakeup request was issued as a result of issuing wup_tsk. | E_OK |
| A wakeup request was issued as a result of issuing iwup_tsk. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |
| Polling failure or timeout. | E_TMOUT |

The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ER      ercd;               /*Declares variable*/
    TMO     tmout = 3600;       /*Declares and initializes variable*/

    /* ............ */

    ercd = tslp_tsk ( tmout );  /*Put task to sleep (with timeout)*/

    if ( ercd == E_OK ) {
        /* ............ */      /*Normal termination processing*/
    } else if ( ercd == E_RLWAI ) {
        /* ............ */      /*Forced termination processing*/
    } else if ( ercd == E_TMOUT ) {
        /* ............ */      /*Timeout processing*/
    }

    /* ............ */
}
```

Note When TMO_FEVR is specified for wait time *tmout*, processing equivalent to slp_tsk will be executed.

## 4.3 Wakeup Task

A task is woken up by issuing the following service call from the processing program.

- wup_tsk, iwup_tsk
  These service calls cancel the WAITING state (sleeping state) of the task specified by parameter *tskid*.
  As a result, the target task is moved from the sleeping state to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  If the target task is in a state other than the sleeping state when this service call is issued, this service call does not move the state but increments the wakeup request counter (by added 0x1 to the wakeup request counter).
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ID      tskid = ID_tskA;    /*Declares and initializes variable*/

    /* ............ */

    wup_tsk ( tskid );          /*Wakeup task*/

    /* ............ */
}
```

Note 1 If the target task is moved to the READY state after this service call is issued, this service call also re-queues the task at the end of the ready queue corresponding to the priority of the task.

Note 2 The wakeup request counter managed by the RI78V4 is configured in 7-bit widths. If the number of wakeup requests exceeds the maximum count value 127 as a result of issuing this service call, the counter manipulation processing is therefore not performed but "E_QOVR" is returned.

## 4.4 Cancel Task Wakeup Requests

A wakeup request is cancelled by issuing the following service call from the processing program.

- can_wup, ican_wup
  These service calls cancel all of the wakeup requests queued to the task specified by parameter *tskid* (the wakeup request counter is set to 0x0).
  When this service call is terminated normally, the number of cancelled wakeup requests is returned.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ER_UINT ercd;               /*Declares variable*/
    ID      tskid = ID_tskA;    /*Declares and initializes variable*/

    /* ............ */

    ercd = can_wup ( tskid );   /*Cancel task wakeup requests*/

    if ( ercd >= 0x0 ) {
        /* ............ */      /*Normal termination processing*/
    }

    /* ............ */
}
```

## 4.5 Release Task from Waiting

The WAITING state is forcibly cancelled by issuing the following service call from the processing program.

- rel_wai, irel_wai
  These service calls forcibly cancel the WAITING state of the task specified by parameter *tskid*.
  As a result, the target task unlinked from the wait queue and is moved from the WAITING state to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  "E_RLWAI" is returned from the service call that triggered the move to the WAITING state (slp_tsk, wai_sem, or the like) to the task whose WAITING state is cancelled by this service call.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ID      tskid = ID_tskA;    /*Declares and initializes variable*/

    /* ............ */

    rel_wai ( tskid );          /*Release task from waiting*/

    /* ............ */
}
```

Note 1 If the target task is moved to the READY state after this service call is issued, this service call also re-queues the task at the end of the ready queue corresponding to the priority of the task.

Note 2 This service call does not perform queuing of forced cancellation requests. If the target task is in a state other than the WAITING or WAITING-SUSPENDED state, "E_OBJ" is returned.

## 4.6 Suspend Task

A task is moved to the SUSPENDED state by issuing the following service call from the processing program.

- sus_tsk, isus_tsk
  These service calls add 0x1 to the suspend request counter for the task specified by parameter *tskid*, and then move the target task from the RUNNING state to the SUSPENDED state, from the READY state to the SUSPENDED state, or from the WAITING state to the WAITING-SUSPENDED state.
  If the target task has moved to the SUSPENDED or WAITING-SUSPENDED state when this service call is issued, the counter manipulation processing is not performed but only the suspend request counter increment processing is executed.
  The SUSPENDED state is cancelled in the following cases, and then moved to the READY state.

| SUSPENDED State Cancel Operation | Return Value |
|---|---|
| A cancel request was issued as a result of issuing rsm_tsk. | E_OK |
| A cancel request was issued as a result of issuing irsm_tsk. | E_OK |
| Forced release from suspended (accept frsm_tsk while suspended). | E_OK |
| Forced release from suspended (accept ifrsm_tsk while suspended). | E_OK |

The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ID      tskid = ID_tskA;    /*Declares and initializes variable*/

    /* ............ */

    sus_tsk ( tskid );          /*Suspend task*/

    /* ............ */
}
```

Note 1 If the target task is the invoking task when this service call is issued, it is unlinked from the ready queue and excluded from the RI78V4 scheduling subject.

Note 2 The suspend request counter managed by the RI78V4 is configured in 7-bit widths. If the number of suspend requests exceeds the maximum count value 127 as a result of issuing this service call, the counter manipulation processing is therefore not performed but "E_QOVR" is returned.

## 4.7 Resume Suspended Task

The SUSPENDED state is cancelled by issuing the following service call from the processing program.

- rsm_tsk, irsm_tsk
  This service call subtracts 0x1 from the suspend request counter for the task specified by parameter *tskid*, and then cancels the SUSPENDED state of the target task.
  As a result, the target task is moved from the SUSPENDED state to the READY state, or from the WAITING-SUSPENDED state to the WAITING state.
  If a suspend request is queued (subtraction result is other than 0x0) when this service call is issued, the counter manipulation processing is not performed but only the suspend request counter decrement processing is executed.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ID      tskid = ID_tskA;    /*Declares and initializes variable*/

    /* ............ */

    rsm_tsk ( tskid );          /*Resume suspended task*/

    /* ............ */
}
```

Note 1 If the target task is moved to the READY state after this service call is issued, this service call also re-queues the task at the end of the ready queue corresponding to the priority of the task.

Note 2 This service call does not perform queuing of cancellation requests. If the target task is in a state other than the SUSPENDED or WAITING-SUSPENDED state, "E_OBJ" is therefore returned.

- frsm_tsk, ifrsm_tsk

  These service calls set the suspend request counter for the task specified by parameter *tskid* to 0x1 f, and then forcibly cancel the SUSPENDED state of the target task.

  As a result, the target task is moved from the SUSPENDED state to the READY state, or from the WAITING-SUSPENDED state to the WAITING state.

  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ID      tskid = ID_tskA;    /*Declares and initializes variable*/

    /* ............ */

    frsm_tsk ( tskid );         /*Forcibly resume suspended task*/

    /* ............ */
}
```

Note 1 If the target task is moved to the READY state after this service call is issued, this service call also re-queues the task at the end of the ready queue corresponding to the priority of the task.

Note 2 This service call does not perform queuing of forced cancellation requests. If the target task is in a state other than the SUSPENDED or WAITING-SUSPENDED state, "E_OBJ" is therefore returned.

## 4.8 Delay Task

A task is moved to the delayed state by issuing the following service call from the processing program.

- dly_tsk
  This service call moves the invoking task from the RUNNING state to the WAITING state (delayed state).
  As a result, the invoking task is unlinked from the ready queue and excluded from the RI78V4 scheduling subject.
  The delayed state is cancelled in the following cases, and then moved to the READY state.

| Delayed State Cancel Operation | Return Value |
|---|---|
| Delay time specified by parameter *dlytim* has elapsed. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |

The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ER      ercd;               /*Declares variable*/
    RELTIM  dlytim = 3600;      /*Declares and initializes variable*/

    /* ............ */

    ercd = dly_tsk ( dlytim );  /*Delay task*/

    if ( ercd == E_OK ) {
        /* ............ */      /*Normal termination processing*/
    } else if ( ercd == E_RLWAI ) {
        /* ............ */      /*Forced termination processing*/
    }

    /* ............ */
}
```

# CHAPTER 5 SYNCHRONIZATION AND COMMUNICATION FUNCTIONS

This chapter describes the synchronization and communication functions performed by the RI78V4.

## 5.1 Outline

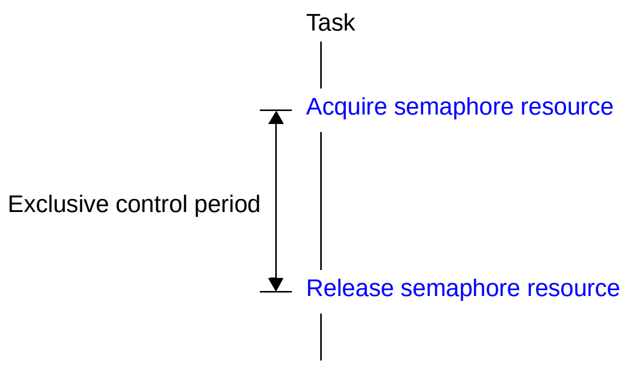
The synchronization and communication functions of the RI78V4 consist of Semaphores, Eventflags, and Mailboxes that are provided as means for realizing exclusive control, queuing, and communication among tasks.

## 5.2 Semaphores

In the RI78V4, non-negative number counting semaphores are provided as a means (exclusive control function) for preventing contention for limited resources (hardware devices, library function, etc.) arising from the required conditions of simultaneously running tasks.

The following shows a processing flow when using a semaphore.

Figure 5-1 Processing Flow (Semaphore)




### 5.2.1 Create semaphore

In the RI78V4, the method of creating a semaphore is limited to "static creation by the Kernel Initialization Module".

Semaphores therefore cannot be created dynamically using a method such as issuing a service call from a processing program.

- Static create
  Static semaphore creation is realized by defining Semaphore information in the system configuration file.
  The RI78V4 executes semaphore creation processing based on data stored in information files, using the Kernel Initialization Module, and handles the created semaphores as management targets.

### 5.2.2 Delete semaphore

In the RI78V4, semaphores created statically by the Kernel Initialization Module cannot be deleted dynamically using a method such as issuing a service call from a processing program.

### 5.2.3 Release semaphore resource

A resource is returned by issuing the following service call from the processing program.

- sig_sem, isig_sem
  These service calls return the resource to the semaphore specified by parameter *semid* (adds 0x1 to the semaphore counter).
  If a task is queued in the wait queue of the target semaphore when this service call is issued, the counter manipulation processing is not performed but the resource is passed to the relevant task (first task of wait queue).
  As a result, the relevant task is unlinked from the wait queue and is moved from the WAITING state (waiting state for a semaphore resource) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ID      semid = ID_semA;    /*Declares and initializes variable*/

    /* ............ */

    sig_sem ( semid );          /*Release semaphore resource*/

    /* ............ */
}
```

Note 1 If the first task linked in the wait queue is moved to the READY state after this service call is issued, this service call also re-queues the task at the end of the ready queue corresponding to the priority of the task.

Note 2 The semaphore counter managed by the RI78V4 is configured in 7-bit widths. If the number of resources exceeds the maximum count value 127 as a result of issuing this service call, the counter manipulation processing is therefore not performed but "E_QOVR" is returned.

### 5.2.4 Acquire semaphore resource

A resource is acquired (waiting forever, polling, or with timeout) by issuing the following service call from the processing program.

- wai_sem
  This service call acquires a resource from the semaphore specified by parameter *semid* (subtracts 0x1 from the semaphore counter).
  If a resource could not be acquired from the target semaphore (semaphore counter is set to 0x0) when this service call is issued, the counter manipulation processing is not performed but the invoking task is queued to the target semaphore wait queue in the order of resource acquisition request (FIFO order).
  As a result, the invoking task is unlinked from the ready queue and is moved from the RUNNING state to the WAITING state (waiting state for a semaphore resource).
  The waiting state for a semaphore state is cancelled in the following cases, and then moved to the READY state.

| Waiting State for a Semaphore State Cancel Operation | Return Value |
|---|---|
| The resource was returned to the target semaphore as a result of issuing sig_sem. | E_OK |
| The resource was returned to the target semaphore as a result of issuing isig_sem. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |

The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ER      ercd;               /*Declares variable*/
    ID      semid = ID_semA;    /*Declares and initializes variable*/

    /* ............ */

    ercd = wai_sem ( semid );   /*Acquire semaphore resource (waiting forever)*/

    if ( ercd == E_OK ) {
        /* ............ */      /*Normal termination processing*/
    } else if ( ercd == E_RLWAI ) {
        /* ............ */      /*Forced termination processing*/
    }

    /* ............ */
}
```

- pol_sem
  This service call acquires a resource from the semaphore specified by parameter *semid* (subtracts 0x1 from the semaphore counter).
  If a resource could not be acquired from the target semaphore (semaphore counter is set to 0x0) when this service call is issued, the counter manipulation processing is not performed but "E_TMOUT" is returned.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ER      ercd;               /*Declares variable*/
    ID      semid = ID_semA;    /*Declares and initializes variable*/

    /* ............ */

    ercd = pol_sem ( semid );   /*Acquire semaphore resource (polling)*/

    if ( ercd == E_OK ) {
        /* ............ */      /*Polling success processing*/
    } else if ( ercd == E_TMOUT ) {
        /* ............ */      /*Polling failure processing*/
    }

    /* ............ */
}
```

- twai_sem
  This service call acquires a resource from the semaphore specified by parameter *semid* (subtracts 0x1 from the semaphore counter).
  If a resource could not be acquired from the target semaphore (semaphore counter is set to 0x0) when this service call is issued, the counter manipulation processing is not performed but the invoking task is queued to the target semaphore wait queue in the order of resource acquisition request (FIFO order).
  As a result, the invoking task is unlinked from the ready queue and is moved from the RUNNING state to the WAITING state (waiting state for a semaphore resource).
  The waiting state for a semaphore resource is cancelled in the following cases, and then moved to the READY state.

| Waiting State for a Semaphore Resource Cancel Operation | Return Value |
|---|---|
| The resource was returned to the target semaphore as a result of issuing sig_sem. | E_OK |
| The resource was returned to the target semaphore as a result of issuing isig_sem. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |
| Polling failure or timeout. | E_TMOUT |

The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ER      ercd;               /*Declares variable*/
    ID      semid = ID_semA;    /*Declares and initializes variable*/
    TMO     tmout = 3600;       /*Declares and initializes variable*/

    /* ............ */

                                /*Acquire semaphore resource (with timeout)*/
    ercd = twai_sem ( semid, tmout );

    if ( ercd == E_OK ) {
        /* ............ */      /*Normal termination processing*/
    } else if ( ercd == E_RLWAI ) {
        /* ............ */      /*Forced termination processing*/
    } else if ( ercd == E_TMOUT ) {
        /* ............ */      /*Timeout processing*/
    }

    /* ............ */
}
```

Note When TMO_FEVR is specified for wait time *tmout*, processing equivalent to wai_sem will be executed. When TMO_POL is specified, processing equivalent to pol_sem will be executed.

### 5.2.5 Reference semaphore state

A semaphore status is referenced by issuing the following service call from the processing program.

- ref_sem
  Stores semaphore state packet (such as existence of waiting tasks) of the semaphore specified by parameter *semid* in the area specified by parameter *pk_rsem*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ID      semid = ID_semA;    /*Declares and initializes variable*/
    T_RSEM  pk_rsem;            /*Declares data structure*/
    ID      wtskid;             /*Declares variable*/
    UINT    semcnt;             /*Declares variable*/

    /* ............ */

    ref_sem ( semid, &pk_rsem );/*Reference semaphore state*/

    wtskid = pk_rsem.wtskid;    /*Reference ID number of the task at the head of
                                  the wait queue*/
    semcnt = pk_rsem.semcnt;    /*Reference current resource count*/

    /* ............ */
}
```

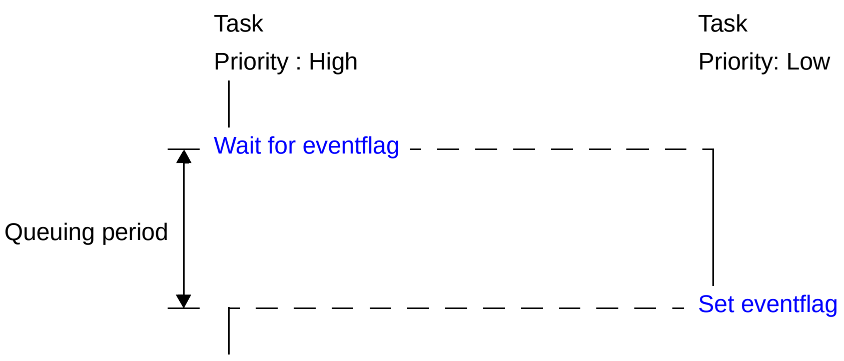
Note For details about the semaphore state packet, refer to "12.5.2 Semaphore state packet".

## 5.3 Eventflags

The RI78V4 provides 16-bit eventflags as a queuing function for tasks, such as keeping the tasks waiting for execution, until the results of the execution of a given processing program are output.

The following shows a processing flow when using an eventflag.

Figure 5-2 Processing Flow (Eventflag)




### 5.3.1 Create eventflag

In the RI78V4, the method of creating an eventflag is limited to "static creation by the Kernel Initialization Module".

Eventflags therefore cannot be created dynamically using a method such as issuing a service call from a processing program.

- Static create

  Static eventflag creation is realized by defining Eventflag information in the system configuration file.

  The RI78V4 executes eventflag creation processing based on data stored in information files, using the Kernel Initialization Module, and handles the created eventflags as management targets.

  Note In the RI78V4, "0x0" is the initial bit pattern for eventflag creation processing.

### 5.3.2 Delete eventflag

In the RI78V4, eventflags created statically by the Kernel Initialization Module cannot be deleted dynamically using a method such as issuing a service call from a processing program.

### 5.3.3 Set eventflag

A bit pattern is set by issuing the following service call from the processing program.

- set_flg, iset_flg
  These service calls set the result of ORing the bit pattern of the eventflag specified by parameter *flgid* and the bit pattern specified by parameter *setptn* as the bit pattern of the target eventflag.
  If the required condition of the task queued to the target eventflag wait queue is satisfied when this service call is issued, the relevant task is unlinked from the wait queue at the same time as bit pattern setting processing.
  As a result, the relevant task is moved from the WAITING state (waiting state for an eventflag) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ID      flgid = ID_flgA;    /*Declares and initializes variable*/
    FLGPTN  setptn = 0B1010;    /*Declares and initializes variable*/

    /* ............ */

    set_flg ( flgid, setptn );  /*Set eventflag*/

    /* ............ */
}
```

Note 1 If the task linked in the wait queue is moved to the READY state after this service call is issued, this service call also re-queues the task at the end of the ready queue corresponding to the priority of the task.

Note 2 If the bit pattern set to the target eventflag is B'1100 and the bit pattern specified by parameter *setptn* is B'1010 when this service call is issued, the bit pattern of the target eventflag is set to B'1110.

### 5.3.4 Clear eventflag

A bit pattern is cleared by issuing the following service call from the processing program.

- clr_flg
  This service call sets the result of ANDing the bit pattern set to the eventflag specified by parameter *flgid* and the bit pattern specified by parameter *clrptn* as the bit pattern of the target eventflag.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ID      flgid = ID_flgA;    /*Declares and initializes variable*/
    FLGPTN  clrptn = 0B1010;    /*Declares and initializes variable*/

    /* ............ */

    clr_flg ( flgid, clrptn );  /*Clear eventflag*/

    /* ............ */
}
```

Note 1 This service call does not perform queuing of clear requests. If the bit pattern has been cleared, therefore, no processing is performed but it is not handled as an error.

Note 2 If the bit pattern set to the target eventflag is B'1100 and the bit pattern specified by parameter *clrptn* is B'1010 when this service call is issued, the bit pattern of the target eventflag is set to B'1110.

Note 3 This service call does not cancel tasks in the waiting state for an eventflag.

### 5.3.5 Wait for eventflag

A bit pattern is checked (waiting forever, polling, or with timeout) by issuing the following service call from the processing program.

- wai_flg
  This service call checks whether the bit pattern specified by parameter *waiptn* and the bit pattern that satisfies the required condition specified by parameter *wfmode* are set to the eventflag specified by parameter *flgid*.
  If a bit pattern that satisfies the required condition has been set for the target eventflag, the bit pattern of the target eventflag is stored in the area specified by parameter *p_flgptn*.
  If the bit pattern of the target eventflag does not satisfy the required condition when this service call is issued, the invoking task is queued to the target eventflag wait queue.
  As a result, the invoking task is unlinked from the ready queue and is moved from the RUNNING state to the WAITING state (waiting state for an eventflag).
  The waiting state for an eventflag is cancelled in the following cases, and then moved to the READY state.

| Waiting State for an Eventflag Cancel Operation | Return Value |
|---|---|
| A bit pattern that satisfies the required condition was set to the target eventflag as a result of issuing set_flg. | E_OK |
| A bit pattern that satisfies the required condition was set to the target eventflag as a result of issuing iset_flg. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |

The following shows the specification format of required condition *wfmode*.

- *wfmode* = TWF_ANDW
  Checks whether all of the bits to which 1 is set by parameter *waiptn* are set as the target eventflag.

- *wfmode* = TWF_ORW
  Checks which bit, among bits to which 1 is set by parameter *waiptn*, is set as the target eventflag.

The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ER      ercd;               /*Declares variable*/
    ID      flgid = ID_flgA;    /*Declares and initializes variable*/
    FLGPTN  waiptn = 0B1110;    /*Declares and initializes variable*/
    MODE    wfmode = TWF_ANDW;  /*Declares and initializes variable*/
    FLGPTN  p_flgptn;           /*Declares variable*/

    /* ............ */

                                /*Wait for eventflag (waiting forever)*/
    ercd = wai_flg ( flgid, waiptn, wfmode, &p_flgptn );

    if ( ercd == E_OK ) {
        /* ............ */      /*Normal termination processing*/
    } else if ( ercd == E_RLWAI ) {
        /* ............ */      /*Forced termination processing*/
```

```
    }

    /* ............ */
}
```

Note 1 In the RI78V4, the number of tasks that can be queued to the eventflag wait queue is one. If this service call is issued for the eventflag to which a task is queued, therefore, "E_ILUSE" is returned regardless of whether or not the required condition is immediately satisfied.

Note 2 The RI78V4 performs bit pattern clear processing (0x0 setting) when the required condition of the target eventflag (TA_CLR attribute) is satisfied.

- pol_flg

  This service call checks whether the bit pattern specified by parameter *waiptn* and the bit pattern that satisfies the required condition specified by parameter *wfmode* are set to the eventflag specified by parameter *flgid*.
  If the bit pattern that satisfies the required condition has been set to the target eventflag, the bit pattern of the target eventflag is stored in the area specified by parameter *p_flgptn*.
  If the bit pattern of the target eventflag does not satisfy the required condition when this service call is issued, "E_TMOUT" is returned.
  The following shows the specification format of required condition *wfmode*.

  - *wfmode* = TWF_ANDW
    Checks whether all of the bits to which 1 is set by parameter *waiptn* are set as the target eventflag.

  - *wfmode* = TWF_ORW
    Checks which bit, among bits to which 1 is set by parameter *waiptn*, is set as the target eventflag.

  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ER      ercd;               /*Declares variable*/
    ID      flgid = ID_flgA;    /*Declares and initializes variable*/
    FLGPTN  waiptn = 0B1110;    /*Declares and initializes variable*/
    MODE    wfmode = TWF_ANDW;  /*Declares and initializes variable*/
    FLGPTN  p_flgptn;           /*Declares variable*/

    /* ............ */

                                /*Wait for eventflag (polling)*/
    ercd = pol_flg ( flgid, waiptn, wfmode, &p_flgptn );

    if ( ercd == E_OK ) {
        /* ............ */      /*Polling success processing*/
    } else if ( ercd == E_TMOUT ) {
        /* ............ */      /*Polling failure processing*/
    }

    /* ............ */
}
```

Note 1 In the RI78V4, the number of tasks that can be queued to the eventflag wait queue is one. If this service call is issued for the eventflag to which a task is queued, therefore, "E_ILUSE" is returned regardless of whether or not the required condition is immediately satisfied.

Note 2 The RI78V4 performs bit pattern clear processing (0x0 setting) when the required condition of the target eventflag (TA_CLR attribute) is satisfied.

- twai_flg

  This service call checks whether the bit pattern specified by parameter *waiptn* and the bit pattern that satisfies the required condition specified by parameter *wfmode* are set to the eventflag specified by parameter *flgid*.
  If the bit pattern that satisfies the required condition has been set to the target eventflag, the bit pattern of the target eventflag is stored in the area specified by parameter *p_flgptn*.
  If the bit pattern of the target eventflag does not satisfy the required condition when this service call is issued, the invoking task is queued to the target eventflag wait queue.
  As a result, the invoking task is unlinked from the ready queue and is moved from the RUNNING state to the WAITING state (waiting state for an eventflag).
  The waiting state for an eventflag is cancelled in the following cases, and then moved to the READY state.

| Waiting State for an Eventflag Cancel Operation | Return Value |
|---|---|
| A bit pattern that satisfies the required condition was set to the target eventflag as a result of issuing set_flg. | E_OK |
| A bit pattern that satisfies the required condition was set to the target eventflag as a result of issuing iset_flg. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |
| Polling failure or timeout. | E_TMOUT |

The following shows the specification format of required condition *wfmode*.

- *wfmode* = TWF_ANDW
  Checks whether all of the bits to which 1 is set by parameter *waiptn* are set as the target eventflag.

- *wfmode* = TWF_ORW
  Checks which bit, among bits to which 1 is set by parameter *waiptn*, is set as the target eventflag.

The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ER      ercd;               /*Declares variable*/
    ID      flgid = ID_flgA;    /*Declares and initializes variable*/
    FLGPTN  waiptn = 0B1110;    /*Declares and initializes variable*/
    MODE    wfmode = TWF_ANDW;  /*Declares and initializes variable*/
    FLGPTN  p_flgptn;           /*Declares variable*/
    TMO     tmout = 3600;       /*Declares and initializes variable*/

    /* ............ */

                                /*Wait for eventflag (with timeout)*/
    ercd = twai_flg ( flgid, waiptn, wfmode, &p_flgptn, tmout );

    if ( ercd == E_OK ) {
        /* ............ */      /*Normal termination processing*/
    } else if ( ercd == E_RLWAI ) {
        /* ............ */      /*Forced termination processing*/
    } else if ( ercd == E_TMOUT ) {
        /* ............ */      /*Timeout processing*/
    }
```

```
    /* ............ */
}
```

Note 1 In the RI78V4, the number of tasks that can be queued to the eventflag wait queue is one. If this service call is issued for the eventflag to which a task is queued, therefore, "E_ILUSE" is returned regardless of whether or not the required condition is immediately satisfied.

Note 2 The RI78V4 performs bit pattern clear processing (0x0 setting) when the required condition of the target eventflag (TA_CLR attribute) is satisfied.

Note 3 When TMO_FEVR is specified for wait time tmout, processing equivalent to wai_flg will be executed. When TMO_POL is specified, processing equivalent to pol_flg will be executed.

### 5.3.6 Reference eventflag state

An eventflag status is referenced by issuing the following service call from the processing program.

- ref_flg
  Stores eventflag state packet (such as existence of waiting tasks) of the eventflag specified by parameter *flgid* in the area specified by parameter *pk_rflg*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ID      flgid = ID_flgA;    /*Declares and initializes variable*/
    T_RFLG  pk_rflg;            /*Declares data structure*/
    ID      wtskid;             /*Declares variable*/
    FLGPTN  flgptn;             /*Declares variable*/

    /* ............ */

    ref_flg ( flgid, &pk_rflg );/*Reference eventflag state*/

    wtskid = pk_rflg.wtskid;    /*Reference ID number of the task at the head of
                                  the wait queue*/
    flgptn = pk_rflg.flgptn;    /*Reference current bit pattern*/

    /* ............ */
}
```

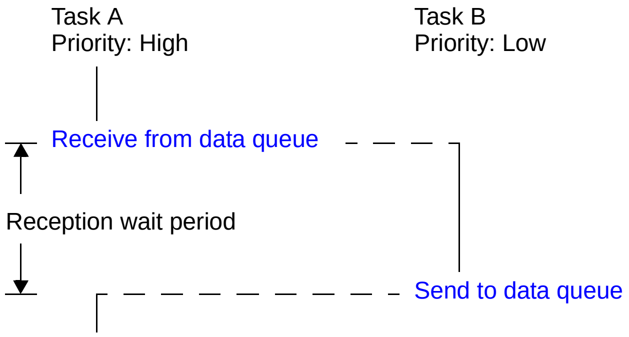
Note For details about the eventflag state packet, refer to "12.5.3 Eventflag state packet".

## 5.4 Data Queues

Multitask processing requires the inter-task communication function (data transfer function) that reports the processing result of a task to another task. The RI78V4 therefore provides the data queues that have the data queue area in which data read/write is enabled for transferring the prescribed size of data.

The following shows a processing flow when using a data queue.

Figure 5-3 Processing Flow (Data Queue)




Note Data units of 4 bytes are transmitted or received at a time.

### 5.4.1 Create data queue

In the RI78V4, the method of creating a deta queue is limited to "static creation".

Data queues therefore cannot be created dynamically using a method such as issuing a service call from a processing program.

Static data queue creation means defining of data queues using static API "CRE_DTQ" in the system configuration file.

For details about the static API "CRE_DTQ", refer to "13.4.4 Data queue information".

### 5.4.2 Send to data queue

A data is transmitted by issuing the following service call from the processing program.

- snd_dtq
  This service call writes data specified by parameter *data* to the data queue area of the data queue specified by parameter *dtqid*.
  If there is no available space for writing data in the data queue area of the target data queue when this service call is issued, this service call does not write data but queues the invoking task to the transmission wait queue of the target data queue and moves it from the RUNNING state to the WAITING state (data transmission wait state).
  The sending WAITING state for a data queue is cancelled in the following cases, and then moved to the READY state.

| Sending WAITING State for a Data Queue Cancel Operation | Return Value |
|---|---|
| Available space was secured in the data queue area of the target data queue as a result of issuing rcv_dtq. | E_OK |
| Available space was secured in the data queue area of the target data queue as a result of issuing prcv_dtq. | E_OK |
| Available space was secured in the data queue area of the target data queue as a result of issuing trcv_dtq. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |

If a task has been queued to the reception wait queue of the target data queue when this service call is issued, this service call does not write data but transfers the data to the task. As a result, the task is unlinked from the reception wait queue and moves from the WAITING state (data reception wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
The following describes an example for coding this service call.

```
#include        <kernel.h>         /*Standard header file definition*/
#include        <kernel_id.h>      /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                  /*Declares variable*/
    ID      dtqid = 1;             /*Declares and initializes variable*/
    VP_INT  data = 123;            /*Declares and initializes variable*/

    /* ......... */

    ercd = snd_dtq (dtqid, data);  /*Send to data queue (waiting forever)*/

    if (ercd == E_OK) {
        /* ......... */            /*Normal termination processing*/
    } else if (ercd == E_RLWAI) {
        /* ......... */            /*Forced termination processing*/
    }

    /* ......... */
}
```

Note 1 Data is written to the data queue area of the target data queue in the order of the data transmission request.

Note 2 Invoking tasks are queued to the transmission wait queue of the target data queue in the order defined during configuration (FIFO order).

- psnd_dtq, ipsnd_dtq

  These service calls write data specified by parameter *data* to the data queue area of the data queue specified by parameter *dtqid*.

  If there is no available space for writing data in the data queue area of the target data queue when either of these service calls is issued, data is not written but E_TMOUT is returned.

  If a task has been queued to the reception wait queue of the target data queue when this service call is issued, this service call does not write data but transfers the data to the task. As a result, the task is unlinked from the reception wait queue and moves from the WAITING state (data reception wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

  The following describes an example for coding this service call.

```
#include        <kernel.h>         /*Standard header file definition*/
#include        <kernel_id.h>      /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                  /*Declares variable*/
    ID      dtqid = 1;             /*Declares and initializes variable*/
    VP_INT  data = 123;            /*Declares and initializes variable*/

    /* ......... */

                                   /*Send to data queue (polling)*/
    ercd = psnd_dtq (dtqid, data);

    if (ercd == E_OK) {
        /* ......... */            /*Polling success processing*/
    } else if (ercd == E_TMOUT) {
        /* ......... */            /*Polling failure processing*/
    }

    /* ......... */
}
```

Note Data is written to the data queue area of the target data queue in the order of the data transmission request.

- tsnd_dtq

  This service call writes data specified by parameter *data* to the data queue area of the data queue specified by parameter *dtqid*.

  If there is no available space for writing data in the data queue area of the target data queue when this service call is issued, the service call does not write data but queues the invoking task to the transmission wait queue of the target data queue and moves it from the RUNNING state to the WAITING state with time (data transmission wait state).

  The sending WAITING state for a data queue is cancelled in the following cases, and then moved to the READY state.

| Sending WAITING State for a Data Queue Cancel Operation | Return Value |
|---|---|
| An available space was secured in the data queue area of the target data queue as a result of issuing rcv_dtq. | E_OK |
| An available space was secured in the data queue area of the target data queue as a result of issuing prcv_dtq. | E_OK |
| An available space was secured in the data queue area of the target data queue as a result of issuing trcv_dtq. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |
| Polling failure or timeout. | E_TMOUT |

If a task has been queued to the reception wait queue of the target data queue when this service call is issued, this service call does not write data but transfers the data to the task. As a result, the task is unlinked from the reception wait queue and moves from the WAITING state (data reception wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

The following describes an example for coding this service call.

```
#include        <kernel.h>        /*Standard header file definition*/
#include        <kernel_id.h>     /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                 /*Declares variable*/
    ID      dtqid = 1;            /*Declares and initializes variable*/
    VP_INT  data = 123;           /*Declares and initializes variable*/
    TMO     tmout = 3600;         /*Declares and initializes variable*/

    /* ......... */

                                  /*Send to data queue (with timeout)*/
    ercd = tsnd_dtq (dtqid, data, tmout);

    if (ercd == E_OK) {
        /* ......... */           /*Normal termination processing*/
    } else if (ercd == E_RLWAI) {
        /* ......... */           /*Forced termination processing*/
    } else if (ercd == E_TMOUT) {
        /* ......... */           /*Timeout processing*/
    }

    /* ......... */
}
```

Note 1 Data is written to the data queue area of the target data queue in the order of the data transmission request.

Note 2 Invoking tasks are queued to the transmission wait queue of the target data queue in the order defined during configuration (FIFO order).

Note 3 TMO_FEVR is specified for wait time *tmout*, processing equivalent to snd_dtq will be executed. When TMO_POL is specified, processing equivalent to psnd_dtq /ipsnd_dtq will be executed.

### 5.4.3 Forced send to data queue

Data is forcibly transmitted by issuing the following service call from the processing program.

- fsnd_dtq, ifsnd_dtq
  These service calls write data specified by parameter *data* to the data queue area of the data queue specified by parameter *dtqid*.
  If there is no available space for writing data in the data queue area of the target data queue when either of these service calls is issued, the service call overwrites data to the area with the oldest data that was written.
  If a task has been queued to the reception wait queue of the target data queue when this service call is issued, this service call does not write data but transfers the data to the task. As a result, the task is unlinked from the reception wait queue and moves from the WAITING state (data reception wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  The following describes an example for coding this service call.

```
#include        <kernel.h>         /*Standard header file definition*/
#include        <kernel_id.h>      /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      dtqid = 1;             /*Declares and initializes variable*/
    VP_INT  data = 123;            /*Declares and initializes variable*/

    /* ......... */

    fsnd_dtq (dtqid, data);        /*Forced send to data queue*/

    /* ......... */
}
```

### 5.4.4 Receive from data queue

A data is received (waiting forever, polling, or with timeout) by issuing the following service call from the processing program.

- rcv_dtq
  This service call reads data in the data queue area of the data queue specified by parameter *dtqid* and stores it to the area specified by parameter *p_data*.
  If no data could be read from the data queue area of the target data queue (no data has been written to the data queue area) when this service call is issued, the service call does not read data but queues the invoking task to the reception wait queue of the target data queue and moves it from the RUNNING state to the WAITING state (data reception wait state).
  The receiving WAITING state for a data queue is cancelled in the following cases, and then moved to the READY state.

| Receiving WAITING State for a Data Queue Cancel Operation | Return Value |
|---|---|
| Data was written to the data queue area of the target data queue as a result of issuing snd_dtq. | E_OK |
| Data was written to the data queue area of the target data queue as a result of issuing psnd_dtq. | E_OK |
| Data was written to the data queue area of the target data queue as a result of issuing ipsnd_dtq. | E_OK |
| Data was written to the data queue area of the target data queue as a result of issuing tsnd_dtq. | E_OK |
| Data was written to the data queue area of the target data queue as a result of issuing fsnd_dtq. | E_OK |
| Data was written to the data queue area of the target data queue as a result of issuing ifsnd_dtq. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |

The following describes an example for coding this service call.

```
#include        <kernel.h>        /*Standard header file definition*/
#include        <kernel_id.h>     /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                 /*Declares variable*/
    ID      dtqid = 1;            /*Declares and initializes variable*/
    VP_INT  p_data;               /*Declares variable*/

    /* ......... */

                                  /*Receive from data queue (waiting forever)*/
    ercd = rcv_dtq (dtqid, &p_data);

    if (ercd == E_OK) {
        /* ......... */           /*Normal termination processing*/
    } else if (ercd == E_RLWAI) {
        /* ......... */           /*Forced termination processing*/
    }

    /* ......... */
}
```

Note 1 Invoking tasks are queued to the reception wait queue of the target data queue in the order of the data reception request.

Note 2 If the receiving WAITING state for a data queue is forcibly released by issuing rel_wai or irel_wai, the contents of the area specified by parameter *p_data* will be undefined.

- prcv_dtq

  These service calls read data in the data queue area of the data queue specified by parameter *dtqid* and stores it to the area specified by parameter *p_data*.

  If no data could be read from the data queue area of the target data queue (no data has been written to the data queue area) when either of these service calls is issued, the service call does not read data but E_TMOUT is returned.

  The following describes an example for coding this service call.

```
#include        <kernel.h>          /*Standard header file definition*/
#include        <kernel_id.h>       /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      dtqid = 1;              /*Declares and initializes variable*/
    VP_INT  p_data;                 /*Declares variable*/

    /* ......... */

                                    /*Receive from data queue (polling)*/
    ercd = prcv_dtq (dtqid, &p_data);

    if (ercd == E_OK) {
        /* ......... */             /*Polling success processing*/
    } else if (ercd == E_TMOUT) {
        /* ......... */             /*Polling failure processing*/
    }

    /* ......... */
}
```

Note If no data could be read from the data queue area of the target data queue (no data has been written to the data queue area) when either of these service calls is issued, the contents in the area specified by parameter *p_data* become undefined.

- trcv_dtq
  This service call reads data in the data queue area of the data queue specified by parameter *dtqid* and stores it to the area specified by parameter *p_data*.
  If no data could be read from the data queue area of the target data queue (no data has been written to the data queue area) when this service call is issued, the service call does not read data but queues the invoking task to the reception wait queue of the target data queue and moves it from the RUNNING state to the WAITING state with time out (data reception wait state).
  The receiving WAITING state for a data queue is cancelled in the following cases, and then moved to the READY state.

| Receiving WAITING State for a Data Queue Cancel Operation | Return Value |
|---|---|
| Data was written to the data queue area of the target data queue as a result of issuing snd_dtq. | E_OK |
| Data was written to the data queue area of the target data queue as a result of issuing psnd_dtq. | E_OK |
| Data was written to the data queue area of the target data queue as a result of issuing ipsnd_dtq. | E_OK |
| Data was written to the data queue area of the target data queue as a result of issuing tsnd_dtq. | E_OK |
| Data was written to the data queue area of the target data queue as a result of issuing fsnd_dtq. | E_OK |
| Data was written to the data queue area of the target data queue as a result of issuing ifsnd_dtq. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |
| Polling failure or timeout. | E_TMOUT |

The following describes an example for coding this service call.

```
#include        <kernel.h>         /*Standard header file definition*/
#include        <kernel_id.h>      /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                  /*Declares variable*/
    ID      dtqid = 1;             /*Declares and initializes variable*/
    VP_INT  p_data;                /*Declares variable*/
    TMO     tmout = 3600;          /*Declares and initializes variable*/

    /* ......... */

                                   /*Receive from data queue (with timeout)*/
    ercd = trcv_dtq (dtqid, &p_data, tmout);

    if (ercd == E_OK) {
            /* ......... */        /*Normal termination processing*/
    } else if (ercd == E_RLWAI) {
            /* ......... */        /*Forced termination processing*/
    } else if (ercd == E_TMOUT) {
            /* ......... */        /*Timeout processing*/
    }

    /* ......... */
}
```

Note 1 Invoking tasks are queued to the reception wait queue of the target data queue in the order of the data reception request.

Note 2 If the data reception wait state is cancelled because rel_wai or irel_wai was issued or the wait time elapsed, the contents in the area specified by parameter *p_data* become undefined.

Note 3 TMO_FEVR is specified for wait time *tmout*, processing equivalent to rcv_dtq will be executed. When TMO_POL is specified, processing equivalent to prcv_dtq will be executed.

### 5.4.5 Reference data queue state

A data queue status is referenced by issuing the following service call from the processing program.

- ref_dtq
  These service calls store the detailed information of the data queue (existence of waiting tasks, number of data elements in the data queue, etc.) specified by parameter *dtqid* into the area specified by parameter *pk_rdtq*.
  The following describes an example for coding this service call.

```
#include        <kernel.h>          /*Standard header file definition*/
#include        <kernel_id.h>       /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      dtqid = 1;              /*Declares and initializes variable*/
    T_RDTQ  pk_rdtq;                /*Declares data structure*/
    ID      stskid;                 /*Declares variable*/
    ID      rtskid;                 /*Declares variable*/
    UINT    sdtqcnt;                /*Declares variable*/
    ATR     dtqatr;                 /*Declares variable*/
    UINT    dtqcnt;                 /*Declares variable*/

    /* ......... */

    ref_dtq (dtqid, &pk_rdtq);      /*Reference data queue state*/

    stskid = pk_rdtq.stskid;        /*Acquires existence of tasks waiting for */
                                    /*data transmission*/
    rtskid = pk_rdtq.rtskid;        /*Acquires existence of tasks waiting for */
                                    /*data reception*/
    sdtqcnt = pk_rdtq.sdtqcnt;      /*Reference the number of data elements in */
                                    /*data queue*/
    dtqatr = pk_rdtq.dtqatr;        /*Reference attribute*/
    dtqcnt = pk_rdtq.dtqcnt;        /*Referene data count*/

    /* ......... */
}
```

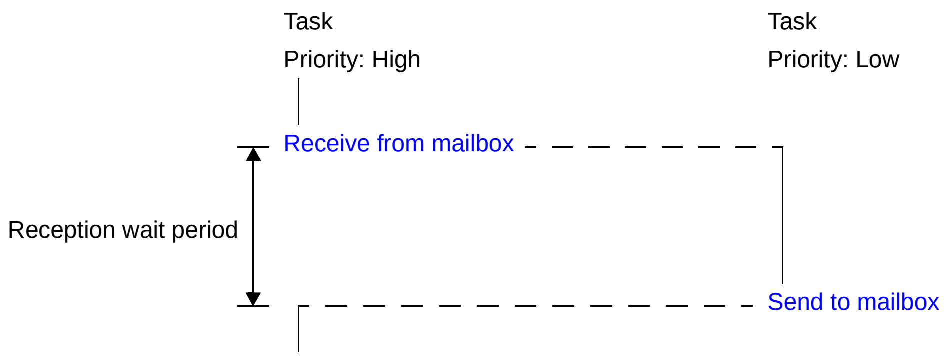
Note For details about the data queue state packet, refer to "12.5.4 Data queue state packet".

## 5.5 Mailboxes

The RI78V4 provides a mailbox, as a communication function between tasks, that hands over the execution result of a given processing program to another processing program.

The following shows a processing flow when using a mailbox.

Figure 5-4 Processing Flow (Mailbox)




### 5.5.1 Create mailbox

In the RI78V4, the method of creating a mailbox is limited to "static creation by the Kernel Initialization Module".

Mailboxes therefore cannot be created dynamically using a method such as issuing a service call from a processing program.

- Static create
  Static mailbox creation is realized by defining Mailbox information in the system configuration file.
  The RI78V4 executes mailbox creation processing based on data stored in information files, using the Kernel Initialization Module, and handles the created mailboxes as management targets.

### 5.5.2 Delete mailbox

In the RI78V4, mailboxes created statically by the Kernel Initialization Module cannot be deleted dynamically using a method such as issuing a service call from a processing program.

### 5.5.3 Message

The information exchanged among processing programs via the mailbox is called "messages".

Messages can be transmitted to any processing program via the mailbox, but it should be noted that, in the case of the synchronization and communication functions of the RI78V4, only the start address of the message is handed over to the receiving processing program, but the message contents are not copied to a separate area.

- Securement of memory area
  In the case of the RI78V4, it is recommended to use the memory area secured by issuing service calls such as get_mpf and pget_mpf for messages.

  Note The RI78V4 uses the message start area as a link area during queuing to the wait queue for mailbox messages. Therefore, if the memory area for messages is secured from other than the memory area controlled by the RI78V4, it must be secured from 4-byte aligned addresses.

- Basic form of messages
  In the RI78V4, the message contents and length are prescribed as follows, according to the attributes of the mailbox to be used.

  - When using a mailbox with the TA_MFIFO attribute
    The contents and length past the first 4 bytes of a message (system reserved area msgque) are not restricted in particular in the RI78V4.
    Therefore, the contents and length past the first 4 bytes are prescribed among the processing programs that exchange data using the mailbox with the TA_MFIFO attribute.
    The following shows the basic form of coding TA_MFIFO attribute messages in C.

    [ Message packet for TA_MFIFO attribute ]

```
typedef struct  t_msg {
    struct  t_msg   __near   *msgque;    /*Reserved for future use*/
} T_MSG;
```

  - When using a mailbox with the TA_MPRI attribute
    The contents and length past the first 5 bytes of a message (system reserved area msgque, priority level msgpri) are not restricted in particular in the RI78V4.
    Therefore, the contents and length past the first 5 bytes are prescribed among the processing programs that exchange data using the mailbox with the TA_MPRI attribute.
    The following shows the basic form of coding TA_MPRI attribute messages in C.

    [ Message packet for TA_MPRI attribute ]

```
typedef struct  t_msg_pri {
    struct  t_msg   __near   *msgque;    /*Reserved for future use*/
    PRI     msgpri;                     /*Message priority*/
} T_MSG_PRI;
```

Note 1 In the RI78V4, a message having a smaller priority number is given a higher priority.

Note 2 A value between 1 and 31 can be specified for message priority.

Note 3 For details about the message packet, refer to "12.5.5 Message packet".

### 5.5.4 Send to mailbox

A message is transmitted by issuing the following service call from the processing program.

- snd_mbx
  This service call transmits the message specified by parameter *pk_msg* to the mailbox specified by parameter *mbxid* (queues the message in the wait queue).
  If a task is queued to the target mailbox wait queue when this service call is issued, the message is not queued but handed over to the relevant task (first task of the wait queue).
  As a result, the relevant task is unlinked from the wait queue and is moved from the WAITING state (receiving waiting state for a mailbox) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ID      mpfid = ID_mpfA;    /*Declares and initializes variable*/
    VP      p_blk;              /*Declares variable*/
    char    *p;                 /*Declares variable*/
    ID      mbxid = ID_mbxA;    /*Declares and initializes variable*/
    T_MSG_PRI       *pk_msg;    /*Declares data structure*/

    /* ............ */

    get_mpf ( mpfid, &p_blk );  /*Secures memory area (for message)*/

                                /*Initializes variable*/
    p = (char *)p_blk + sizeof (T_MSG_PRI);

    while ( expr ) {
            *p++ = ............ /*Creates message (contents)*/
    }

                                /*Initializes data structure*/
    (T_MSG_PRI *)p_blk->msgpri = 8;

                                /*Send to mailbox*/
    snd_mbx ( mbxid, (T_MSG_PRI *)p_blk );

    /* ............ */
}
```

Note 1 If the first task of the wait queue is moved to the READY state after this service call is issued, this service call also re-queues the task at the end of the ready queue corresponding to the priority of the task.

Note 2 Messages are queued to the target mailbox wait queue in the order defined by Attribute (queuing method): mbxatr during configuration (FIFO order or priority order).

Note 3 With the RI78V4 mailbox, only the start address of the message is handed over to the receiving processing program, but the message contents are not copied to a separate area. The message contents can therefore be rewritten even after this service call is issued.

Note 4 For details about the message packet, refer to "12.5.5 Message packet".

### 5.5.5 Receive from mailbox

A message is received (waiting forever, polling, or with timeout) by issuing the following service call from the processing program.

- rcv_mbx
  This service call receives a message from the mailbox specified by parameter *mbxid*, and stores its start address in the area specified by parameter *ppk_msg*.
  If the message could not be received from the target mailbox (no messages were queued in the wait queue) when this service call is issued, message reception processing is not executed but the invoking task is queued to the target mailbox wait queue in the order of message reception request (FIFO order).
  As a result, the invoking task is unlinked from the ready queue and is moved from the RUNNING state to the WAITING state (receiving waiting for a mailbox).
  The receiving waiting for a mailbox is cancelled in the following cases, and then moved to the READY state.

| Receiving Waiting for a Mailbox Cancel Operation | Return Value |
|---|---|
| A message was transmitted to the target mailbox as a result of issuing snd_mbx. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |

The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ER      ercd;               /*Declares variable*/
    ID      mbxid = ID_mbxA;    /*Declares and initializes variable*/
    T_MSG   *ppk_msg;           /*Declares data structure*/

    /* ............ */

                                /*Receive from mailbox (waiting forever)*/
    ercd = rcv_mbx ( mbxid, &ppk_msg );

    if ( ercd == E_OK ) {
        /* ............ */      /*Normal termination processing*/
    } else if ( ercd == E_RLWAI ) {
        /* ............ */      /*Forced termination processing*/
    }

    /* ............ */
}
```

Note For details about the message packet, refer to "12.5.5 Message packet".

- prcv_mbx
  This service call receives a message from the mailbox specified by parameter *mbxid*, and stores its start address in the area specified by parameter *ppk_msg*.
  If the message could not be received from the target mailbox (no messages were queued in the wait queue) when this service call is issued, message reception processing is not executed but "E_TMOUT" is returned.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ER      ercd;               /*Declares variable*/
    ID      mbxid = ID_mbxA;    /*Declares and initializes variable*/
    T_MSG   *ppk_msg;           /*Declares data structure*/

    /* ............ */

                                /*Receive from mailbox (polling)*/
    ercd = prcv_mbx ( mbxid, &ppk_msg );

    if ( ercd == E_OK ) {
        /* ............ */      /*Polling success processing*/
    } else if ( ercd == E_TMOUT ) {
        /* ............ */      /*Polling failure processing*/
    }

    /* ............ */
}
```

Note    For details about the message packet, refer to "12.5.5 Message packet".

- trcv_mbx
  This service call receives a message from the mailbox specified by parameter *mbxid*, and stores its start address in the area specified by parameter *ppk_msg*.
  If the message could not be received from the target mailbox (no messages were queued in the wait queue) when this service call is issued, message reception processing is not executed but the invoking task is queued to the target mailbox wait queue in the order of message reception request (FIFO order).
  As a result, the invoking task is unlinked from the ready queue and is moved from the RUNNING state to the WAITING state (receiving waiting for a mailbox).
  The receiving waiting for a mailbox is cancelled in the following cases, and then moved to the READY state.

| Receiving Waiting for a Mailbox Cancel Operation | Return Value |
|---|---|
| A message was transmitted to the target mailbox as a result of issuing snd_mbx. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |
| Polling failure or timeout. | E_TMOUT |

The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ER      ercd;               /*Declares variable*/
    ID      mbxid = ID_mbxA;    /*Declares and initializes variable*/
    T_MSG   *ppk_msg;           /*Declares data structure*/
    TMO     tmout = 3600;       /*Declares and initializes variable*/

    /* ............ */

                                /*Receive from mailbox (with timeout)*/
    ercd = trcv_mbx ( mbxid, &ppk_msg, tmout );

    if ( ercd == E_OK ) {
        /* ............ */      /*Normal termination processing*/
    } else if ( ercd == E_RLWAI ) {
        /* ............ */      /*Forced termination processing*/
    } else if ( ercd == E_TMOUT ) {
        /* ............ */      /*Timeout processing*/
    }

    /* ............ */
}
```

Note 1 When TMO_FEVR is specified for wait time *tmout*, processing equivalent to rcv_mbx will be executed. When TMO_POL is specified, processing equivalent to prcv_mbx will be executed.

Note 2 For details about the message packet, refer to "12.5.5 Message packet".

### 5.5.6 Reference mailbox state

A mailbox status is referenced by issuing the following service call from the processing program.

- ref_mbx
  Stores mailbox state packet (such as existence of waiting tasks) of the mailbox specified by parameter *mbxid* in the area specified by parameter *pk_rmbx*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ID      mbxid = ID_mbxA;    /*Declares and initializes variable*/
    T_RMBX  pk_rmbx;            /*Declares data structure*/
    ID      wtskid;             /*Declares variable*/
    T_MSG   *pk_msg;            /*Declares data structure*/

    /* ............ */

    ref_mbx ( mbxid, &pk_rmbx );/*Reference mailbox state*/

    wtskid = pk_rmbx.wtskid;    /*Reference ID number of the task at the head of
                                  the wait queue*/
    pk_msg = pk_rmbx.pk_msg;    /*Referenc start address of the message packet at
                                  the head of the message queue*/

    /* ............ */
}
```

Note For details about the mailbox state packet, refer to "12.5.6 Mailbox state packet".

# CHAPTER 6 MEMORY POOL MANAGEMENT FUNCTIONS

This chapter describes the memory pool management functions performed by the RI78V4.

## 6.1 Outline

The statically secured memory areas in the Kernel Initialization Module are subject to management by the memory pool management functions of the RI78V4.

In the RI78V4, the allocation destinations (section names) of management objects modularized for each function are specified.

The following lists the section names prescribed in the RI78V4.

- .kernel_system section
  Area where the RI78V4's core processing part and main processing part of service calls provided by the RI78V4 are to be allocated.

- .kernel_system_timer_n section
  Area where the interrupt fo system timer and information of FAR branch are to be allocated.

- .kernel_info section
  Area where information items such as the RI78V4 version are to be allocated.

- .kernel_const / .kernel_const_f section
  Area where initial information items related to OS resources that do not change dynamically are allocated as system information tables and interrupt information definition file.

- .kernel_stack section
  Area where the system stack and the task stack are to be allocated.

- .kernel_data section
  Area where information items required to implement the functionalities provided by the RI78V4 and information items related to OS resources that change dynamically are allocated as management objects.

- .kernel_data_init section
  Area where initial information items of RI78V4 are to be allocated.

- .kernel_work0, .kernel_work1, .kernel_work2, .kernel_work3 section
  Area where data of data queue and fixed-sized memory pools are to be allocated.

- .kernel_data_trace_n section
  Area where the trace data are to be allocated.

- .kernel_const_trace_f section
  Area where information items to get trace data are to be allocated.

- .kernel_system_trace_f section
  Area where the processing part to get trace data are to be allocated.

- .kernel_sbss section
  SADDR area where the RI78V4's core processing are to be allocated.

## 6.2 Fixed-Sized Memory Pool

When a dynamic memory manipulation request is issued from a processing program in the RI78V4, the fixed-sized memory pool is provided as a usable memory area.

Dynamic memory manipulation of the fixed-sized memory pool is executed in fixed size memory block units.

### 6.2.1 Create fixed-sized memory pool

In the RI78V4, the method of creating a fixed-sized memory pool is limited to "static creation by the Kernel Initialization Module".

Fixed-sized memory pools therefore cannot be created dynamically using a method such as issuing a service call from a processing program.

- Static create
  Static fixed-sized memory pool creation is realized by defining Fixed-sized memory pool information in the system configuration file.
  The RI78V4 executes fixed-sized memory pool creation processing based on data stored in information files, using the Kernel Initialization Module, and handles the created fixed-sized memory pools as management targets.

### 6.2.2 Delete fixed-sized memory pool

In the RI78V4, fixed-sized memory pools created statically by the Kernel Initialization Module cannot be deleted dynamically using a method such as issuing a service call from a processing program.

### 6.2.3 Acquire fixed-sized memory block

A memory block is acquired (waiting forever, polling, or with timeout) by issuing the following service call from the processing program.

- get_mpf
  This service call acquires the memory block from the fixed-sized memory pool specified by parameter *mpfid* and stores the start address in the area specified by parameter *p_blk*.
  If a memory block could not be acquired from the target fixed-sized memory pool (no available memory blocks exist) when this service call is issued, memory block acquisition processing is not performed but the invoking task is queued to the target fixed-sized memory pool wait queue in the order of memory block acquisition request (FIFO order).
  As a result, the invoking task is unlinked from the ready queue and is moved from the RUNNING state to the WAITING state (waiting state for a fixed-sized memory block).
  The waiting state for a fixed-sized memory block is cancelled in the following cases, and then moved to the READY state.

| Waiting State for a Fixed-sized Memory Block Cancel Operation | Return Value |
|---|---|
| A memory block was returned to the target fixed-sized memory pool as a result of issuing rel_mpf. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |

The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ER      ercd;               /*Declares variable*/
    ID      mpfid = ID_mpfA;    /*Declares and initializes variable*/
    VP      p_blk;              /*Declares variable*/

    /* ............ */

                                /*Acquire fixed-sized memory block (wait
                                  forever)*/
    ercd = get_mpf ( mpfid, &p_blk );

    if ( ercd == E_OK ) {
        /* ............ */      /*Normal termination processing*/

                                /*Release fixed-sized memory block*/
        rel_mpf ( mpfid, p_blk );
    } else if ( ercd == E_RLWAI ) {
        /* ............ */      /*Forced termination processing*/
    }

    /* ............ */
}
```

- pget_mpf

This service call acquires the memory block from the fixed-sized memory pool specified by parameter *mpfid* and stores the start address in the area specified by parameter *p_blk*.
If a memory block could not be acquired from the target fixed-sized memory pool (no available memory blocks exist) when this service call is issued, memory block acquisition processing is not performed but "E_TMOUT" is returned.
The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ER      ercd;               /*Declares variable*/
    ID      mpfid = ID_mpfA;    /*Declares and initializes variable*/
    VP      p_blk;              /*Declares variable*/

    /* ............ */

                                /*Acquire fixed-sized memory block (polling)*/
    ercd = pget_mpf ( mpfid, &p_blk );

    if ( ercd == E_OK ) {
        /* ............ */      /*Polling success processing*/

                                /*Release fixed-sized memory block*/
        rel_mpf ( mpfid, p_blk );
    } else if ( ercd == E_TMOUT ) {
        /* ............ */      /*Polling failure processing*/
    }

    /* ............ */
}
```

- tget_mpf
  This service call acquires the memory block from the fixed-sized memory pool specified by parameter *mpfid* and stores the start address in the area specified by parameter *p_blk*.
  If a memory block could not be acquired from the target fixed-sized memory pool (no available memory blocks exist) when this service call is issued, memory block acquisition processing is not performed but the invoking task is queued to the target fixed-sized memory pool wait queue in the order of memory block acquisition request (FIFO order).
  As a result, the invoking task is unlinked from the ready queue and is moved from the RUNNING state to the WAITING state (waiting state for a fixed-sized memory block).
  The waiting state for a fixed-sized memory block is cancelled in the following cases, and then moved to the READY state.

| Waiting State for a Fixed-sized memory Block Cancel Operation | Return Value |
|---|---|
| A memory block was returned to the target fixed-sized memory pool as a result of issuing rel_mpf. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |
| Polling failure or timeout. | E_TMOUT |

The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ER      ercd;               /*Declares variable*/
    ID      mpfid = ID_mpfA;    /*Declares and initializes variable*/
    VP      p_blk;              /*Declares variable*/
    TMO     tmout = 3600;       /*Declares and initializes variable*/

    /* ............ */

                                /*Acquire fixed-sized memory block (with
                                  timeout)*/
    ercd = tget_mpf ( mpfid, &p_blk, tmout );

    if ( ercd == E_OK ) {
        /* ............ */      /*Normal termination processing*/

                                /*Release fixed-sized memory block*/
        rel_mpf ( mpfid, p_blk );
    } else if ( ercd == E_RLWAI ) {
        /* ............ */      /*Forced termination processing*/
    } else if ( ercd == E_TMOUT ) {
        /* ............ */      /*Timeout processing*/
    }

    /* ............ */
}
```

Note When TMO_FEVR is specified for wait time *tmout*, processing equivalent to get_mpf will be executed. When TMO_POL is specified, processing equivalent to pget_mpf will be executed.

### 6.2.4 Release fixed-sized memory block

A memory block is returned by issuing the following service call from the processing program.

- rel_mpf

  This service call returns the memory block specified by parameter *blk* to the fixed-sized memory pool specified by parameter *mpfid*.

  If a task is queued to the target fixed-sized memory pool wait queue when this service call is issued, memory block return processing is not performed but memory blocks are returned to the relevant task (first task of wait queue).

  As a result, the relevant task is unlinked from the wait queue and is moved from the WAITING state (waiting state for a fixed-sized memory block) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ER      ercd;               /*Declares variable*/
    ID      mpfid = ID_mpfA;    /*Declares and initializes variable*/
    VP      blk;                /*Declares variable*/

    /* ............ */

                                /*Acquire fixed-sized memory block*/
    ercd = get_mpf ( mpfid, &blk );

    if ( ercd == E_OK ) {
        /* ............ */      /*Normal termination processing*/

                                /*Release fixed-sized memory block*/
        rel_mpf ( mpfid, blk );
    } else if ( ercd == E_RLWAI ) {
        /* ............ */      /*Forced termination processing*/
    }

    /* ............ */
}
```

Note 1 If the first task of the wait queue is moved to the READY state after this service call is issued, this service call also re-queues the task at the end of the ready queue corresponding to the priority of the task.

Note 2 The RI78V4 does not clear the memory blocks before returning them. The contents of the returned memory blocks are therefore undefined.

### 6.2.5 Reference fixed-sized memory pool state

A fixed-sized memory pool status is referenced by issuing the following service call from the processing program.

- ref_mpf
  Stores fixed-sized memory pool state packet (such as existence of waiting tasks) of the fixed-sized memory pool specified by parameter *mpfid* in the area specified by parameter *pk_rmpf*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ID      mpfid = ID_mpfA;    /*Declares and initializes variable*/
    T_RMPF  pk_rmpf;            /*Declares data structure*/
    ID      wtskid;             /*Declares variable*/
    UINT    fblkcnt;            /*Declares variable*/

    /* ............ */

    ref_mpf ( mpfid, &pk_rmpf );/*Reference fixed-sized memory pool state*/

    wtskid = pk_rmpf.wtskid;    /*Reference ID number of the task at the head of
                                  the wait queue*/
    fblkcnt = pk_rmpf.fblkcnt;  /*Reference number of free memory blocks*/

    /* ............ */
}
```

Note For details about the fixed-sized memory pool state packet, refer to "12.5.7 Fixed-sized memory pool state packet".

# CHAPTER 7 TIME MANAGEMENT FUNCTIONS

This chapter describes the time management functions performed by the RI78V4.

## 7.1 Outline

The time management functions of the RI78V4 include Delayed Wakeup, Timeout, and Cyclic Handlers that use timer interrupts created as fixed intervals, as means for realizing time-dependent processing.

Note The RI78V4 does not execute initialization of hardware that creates timer interrupts (clock controller, etc.). This initialization processing must therefore be coded by the user in the Boot Processing or Initialization Routine.

## 7.2 Timer Handler

The timer handler is a dedicated time control processing routine that consists of the processing required to realize delayed wakeup of tasks, timeout during the WAITING state, and cyclic handler activation, and is called from the interrupt handler that is activated upon output of a timer interrupt.

Note The timer handler is part of the functions provided by the RI78V4. The user therefore need not code the processing contents of the timer handler.

### 7.2.1 Define timer handler

Timer handler registration is registerd by CF78V4 based on the clock timer interrupt source in system configuration file. So it is not necessory to discribe the timer handler by user.

## 7.3 Delayed Wakeup

Delayed wakeup the operation that makes the invoking task transit from the RUNNING state to the WAITING state during the interval until a given length of time has elapsed, and makes that task move from the WAITING state to the READY state once the given length of time has elapsed.

Delayed wakeup is implemented by issuing the following service call from the processing program.

Table 7-1 Delayed Wakeup

| Service Call | Function |
|---|---|
| dly_tsk | Delay task. |

## 7.4 Timeout

Timeout is the operation that makes the target task move from the RUNNING state to the WAITING state during the interval until a given length of time has elapsed if the required condition issued from a task is not immediately satisfied, and makes that task move from the WAITING state to the READY state regardless of whether the required condition is satisfied once the given length of time has elapsed.

A timeout is implemented by issuing the following service call from the processing program.

Table 7-2 Timeout

| Service Call | Function |
|---|---|
| tslp_tsk | Put task to sleep. |
| twai_sem | Acquire semaphore resource. |
| twai_flg | Wait for eventflag. |
| tsnd_dtq | Send to data queue. |
| trcv_dtq | Receive from data queue. |
| trcv_mbx | Receive from mailbox. |
| tget_mpf | Acquire fixed-sized memory block. |

## 7.5 Cyclic Handlers

The cyclic handler is a routine dedicated to cycle processing that is activated periodically at a constant interval (activation cycle), and is called from the Timer Handler.

The RI78V4 handles the cyclic handler as a "non-task (module independent from tasks)". Therefore, even if a task with the highest priority in the system is being executed, the processing is suspended when a specified activation cycle has come, and the control is passed to the cyclic handler.

### 7.5.1 Create cyclic handler

In the RI78V4, the method of creating a cyclic handler is limited to "static creation by the Kernel Initialization Module".

Cyclic handlers therefore cannot be created dynamically using a method such as issuing a service call from a processing program.

- Static create
  Static cyclic handler creation is realized by defining Cyclic handler information in the system configuration file.
  The RI78V4 executes cyclic handler creation processing based on data stored in information files, using the Kernel Initialization Module, and handles the created cyclic handlers as management targets.

### 7.5.2 Delete cyclic handler

In the RI78V4, cyclic handlers created statically by the Kernel Initialization Module cannot be deleted dynamically using a method such as issuing a service call from a processing program.

### 7.5.3 Basic form of cyclic handlers

Write cyclic handlers using void type functions that do not have arguments (function: any).

The following shows the basic form of cyclic handlers.

[ C Language ]

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_cychdr ( void )
{
    /* ............ */          /*Main processing*/

    return;                     /*Terminate cyclic handler*/
}
```

[ Assembly Language ]

```
$INCLUDE    (kernel.inc)        ;Standard header file definition
$INCLUDE    (kernel_id.inc)     ;System information header file definition

    .PUBLIC  _func_cychdr
    .SECTION .text, TEXT
_func_cychdr:
    ; ............              ;Main Processing

    RET                         ;Terminate cyclic handler
```

### 7.5.4 Internal processing of cyclic handler

The RI78V4 handles the cyclic handler as a "non-task".

Moreover, the RI78V4 executes "original pre-processing" when passing control to the cyclic handler, as well as "original post-processing" when regaining control from the cyclic handler.

Therefore, note the following points when coding cyclic handlers.

- Coding method
  Code cyclic handlers using C or assembly language in the format shown in "7.5.3 Basic form of cyclic handlers".

- Stack switching
  The RI78V4 executes processing to switch to the system stack when passing control to the cyclic handler, and processing to switch to the stack for the switch destination processing program (system stack or task stack) when regaining control from the cyclic handler.
  The user is therefore not required to code processing related to stack switching in cyclic handlers.

- Interrupt status
  Maskable interrupt acknowledgement is prohibited in the RI78V4 when control is passed to the cyclic handler.
  To change (enable) the interrupt status in the cyclic handler, calling of the __EI function are therefore required.

- Service call issuance
  The RI78V4 handles the cyclic handler as a "non-task".
  Service calls that can be issued in cyclic handlers are limited to the service calls that can be issued from non-tasks.

  Note 1 For details on the valid issuance range of each service call, refer to Table 12-8 to Table 12-17.

  Note 2 If a service call (ichg_pri, isig_sem, etc.) accompanying dispatch processing (task scheduling processing) is issued in order to quickly complete the processing in the cyclic handler during the interval until the processing in the cyclic handler ends, the RI78V4 executes only processing such as queue manipulation, counter manipulation, etc., and the actual dispatch processing is delayed until a return instruction is issued by the cyclic handler, upon which the actual dispatch processing is performed in batch.

### 7.5.5 Start cyclic handler operation

Moving to the operational state (STA state) is implemented by issuing the following service call from the processing program.

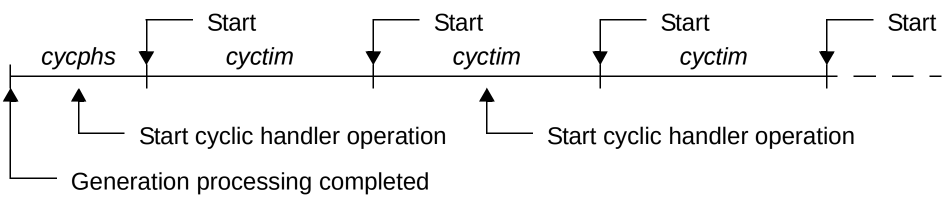
- sta_cyc
  This service call moves the cyclic handler specified by parameter *cycid* from the non-operational state (STP state) to operational state (STA state).
  As a result, the target cyclic handler is handled as an activation target of the RI78V4.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ID      cycid = ID_cycA;    /*Declares and initializes variable*/

    /* ............ */

    sta_cyc ( cycid );          /*Start cyclic handler operation*/

    /* ............ */
}
```

Note The relative interval from when either of this service call is issued until the first activation request is issued varies depending on whether the TA_PHS attribute is specified for the target cyclic handler during configuration.

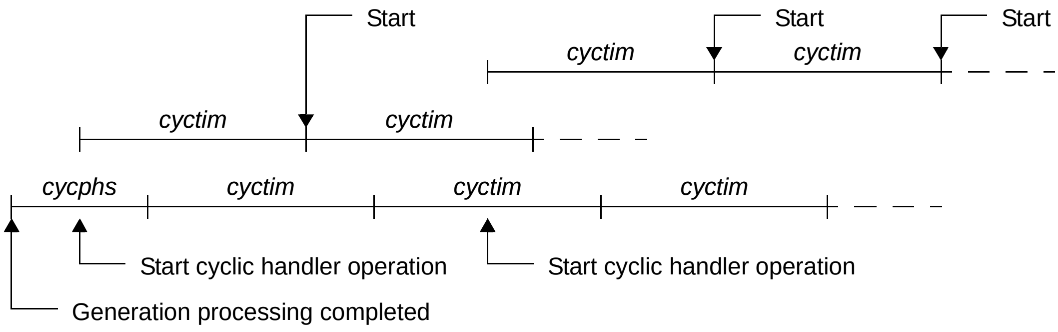
[ Cyclic handler activation image(the TA_PHS attribute is specified) ]

The target cyclic handler activation timing is set based on the activation phases (initial activation phase *cycphs* and activation cycle *cyctim*) defined during configuration.
If the target cyclic handler has already been started, however, no processing is performed even if this service call is issued, but it is not handled as an error.
The following shows a cyclic handler activation timing image.




[ Cyclic handler activation image(the TA_PHS attribute is not specified) ]

The target cyclic handler activation timing is set based on the activation phase (activation cycle *cyctim*) when this service call is issued.
This setting is performed regardless of the operating status of the target cyclic handler.
The following shows a cyclic handler activation timing image.


Start
Start
Start
cyctim
cyctim
cyctim
cyctim
cycphs
cyctim
cyctim
cyctim
Start cyclic handler operation
Start cyclic handler operation
Generation processing completed

### 7.5.6 Stop cyclic handler operation

Moving to the non-operational state (STP state) is implemented by issuing the following service call from the processing program.

- stp_cyc
  This service call moves the cyclic handler specified by parameter *cycid* from the operational state (STA state) to non-operational state (STP state).
  As a result, the target cyclic handler is excluded from activation targets of the RI78V4 until issuance of sta_cyc.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ID      cycid = ID_cycA;    /*Declares and initializes variable*/

    /* ............ */

    stp_cyc ( cycid );          /*Stop cyclic handler operation*/

    /* ............ */
}
```

Note This service call does not perform queuing of stop requests. If the target cyclic handler has been moved to the non-operational state (STP state), therefore, no processing is performed but it is not handled as an error.

### 7.5.7 Reference cyclic handler state

A cyclic handler status by issuing the following service call from the processing program.

- ref_cyc
  Stores cyclic handler state packet (such as current status) of the cyclic handler specified by parameter *cycid* in the area specified by parameter *pk_rcyc*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ID      cycid = ID_cycA;    /*Declares and initializes variable*/
    T_RCYC  pk_rcyc;            /*Declares data structure*/
    STAT    cycstat;            /*Declares variable*/
    RELTIM  lefttim;            /*Declares variable*/

    /* ............ */

    ref_cyc ( cycid, &pk_rcyc );/*Reference cyclic handler state*/

    cycstat = pk_rcyc.cycstat;  /*Reference cyclic handler operational state*/
    lefttim = pk_rcyc.lefttim;  /*Reference time left before the next activation*/

    /* ............ */
}
```

Note For details about the cyclic handler state packet, refer to "12.5.8 Cyclic handler state packet".

# CHAPTER 8 SYSTEM STATE MANAGEMENT FUNCTIONS

This chapter describes the system state management functions performed by the RI78V4.

## 8.1 Outline

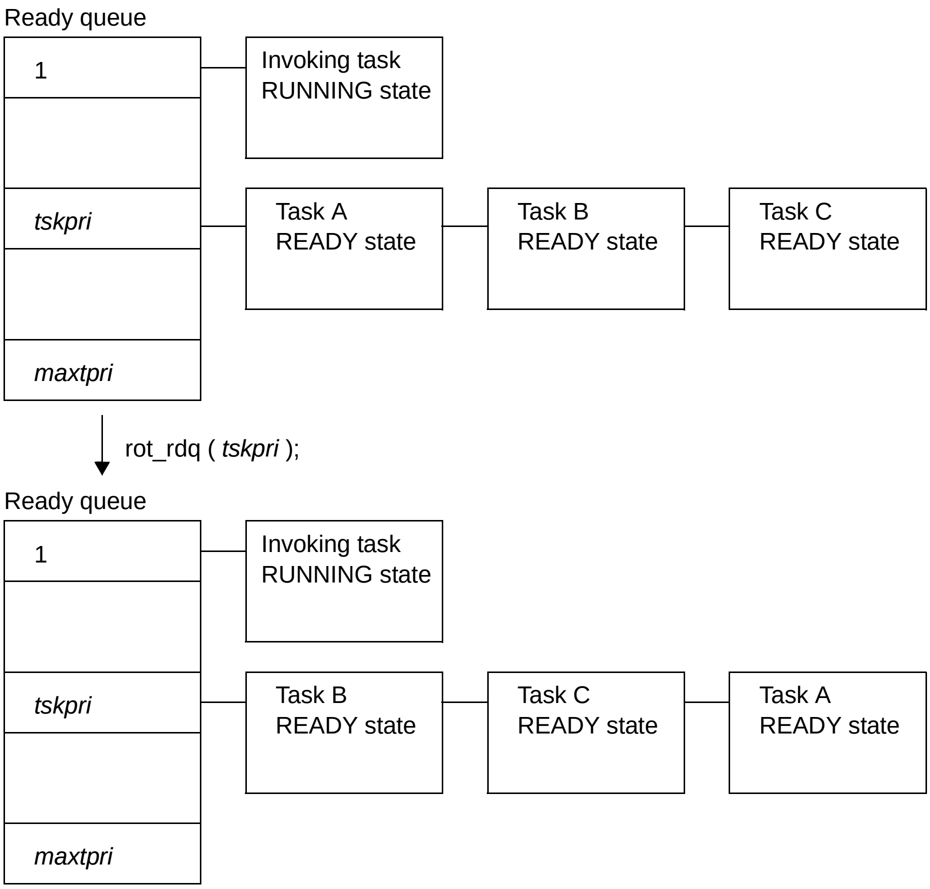
The system state control functions of the RI78V4 include, in addition to functions to manipulate the state of the system, such as transition to the CPU locked state and transition to the dispatching disabled state, functions for referencing the state of the system, such as context type referencing and CPU locked state referencing.

## 8.2 Rotate Task Precedence

A ready queue is rotated by issuing the following service call from the processing program.

- rot_rdq, irot_rdq
  This service call re-queues the first task of the ready queue corresponding to the priority specified by parameter *tskpri* to the end of the queue to change the task execution order explicitly.
  The following shows the status transition when this service call is used.

Figure 8-1 Rotate Task Precedence



The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_cychdr ( void )
{
    PRI     tskpri = 8;         /*Declares and initializes variable*/

    /* ............ */

    irot_rdq ( tskpri );        /*Rotate task precedence*/

    /* ............ */

    return;                     /*Terminate cyclic handler*/
}
```

Note 1 This service call does not perform queuing of rotation requests. If no task is queued to the ready queue corresponding to the relevant priority, therefore, no processing is performed but it is not handled as an error.

Note 2 Round-robin scheduling can be implemented by issuing this service call via a cyclic handler in a constant cycle.

Note 3 The ready queue is a hash table that uses priority as the key, and tasks that have entered an executable state (READY state or RUNNING state) are queued in FIFO order.
Therefore, the scheduler realizes the RI78V4's Scheduling System by executing task detection processing from the highest priority level of the ready queue upon activation, and upon detection of queued tasks, giving the CPU use right to the first task of the proper priority level.

## 8.3 Reference Task ID in the RUNNING State

A RUNNING-state task is referenced by issuing the following service call from the processing program.

- get_tid, iget_tid
  These service calls store the ID of a task in the RUNNING state in the area specified by parameter *p_tskid*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_cychdr ( void )
{
    ID      p_tskid;            /*Declares variable*/

    /* ............ */

    iget_tid ( &p_tskid );      /*Reference task ID in the RUNNING state*/

    /* ............ */

    return;                     /*Terminate cyclic handler*/
}
```

Note This service call stores TSK_NONE in the area specified by parameter *p_tskid* if no tasks that have entered the RUNNING state exist (all tasks in the IDLE state).

## 8.4 Lock the CPU

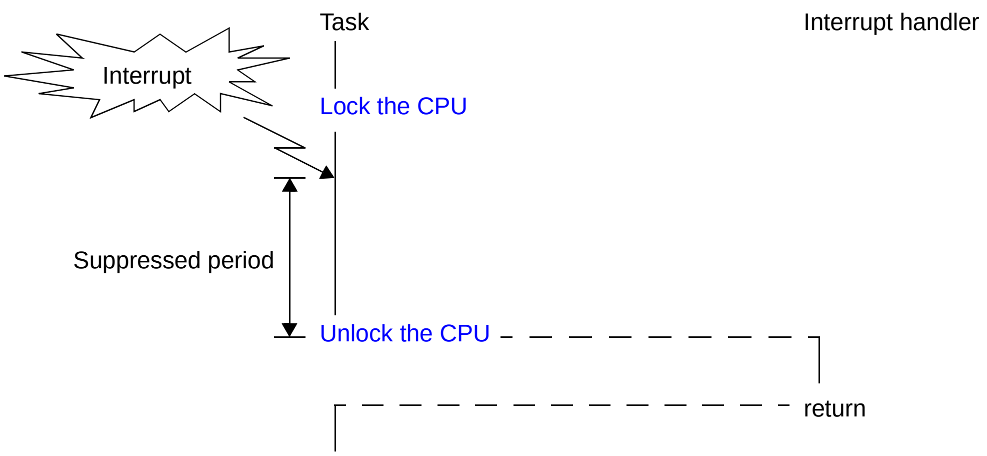
A task is moved to the CPU locked state by issuing the following service call from the processing program.

- loc_cpu, iloc_cpu
  These service calls change the system status type to the CPU locked state.
  As a result, maskable interrupt acknowledgment processing is prohibited during the interval from this service call is issued until unl_cpu or iunl_cpu is issued, and service call issuance is also restricted.
  If a maskable interrupt is created during this period, the RI78V4 delays transition to the relevant interrupt processing (interrupt handler) until either unl_cpu or iunl_cpu is issued.
  The service calls that can be issued in the CPU locked state are limited to the one listed below.

| Service Call | Function |
|---|---|
| loc_cpu, iloc_cpu | Lock the CPU. |
| unl_cpu, iunl_cpu | Unlock the CPU. |
| sns_ctx | Reference contexts. |
| sns_loc | Reference CPU state. |
| sns_dsp | Reference dispatching state. |
| sns_dpn | Reference dispatch pending state. |

The following shows a processing flow when using this service call.

Figure 8-2 Lock the CPU



The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    /* ............ */

    loc_cpu ( );                /*Lock the CPU*/

    /* ............ */          /*CPU locked state*/

    unl_cpu ( );                /*Unlock the CPU*/

    /* ............ */
}
```

Note 1 The CPU locked state changed by issuing this service call must be cancelled before the processing program that issued this service call ends.

Note 2 This service call does not perform queuing of lock requests. If the system is in the CPU locked state, therefore, no processing is performed but it is not handled as an error.

Note 3 The RI78V4 implements disabling of maskable interrupt acknowledgment bu manipulating the interrupt mask flag register (MK*xx*) and the in-service priority flag (ISP*x*) of the program status word (PSW). Therefore, manipulating of these registers from the processing program is prohibited from when this service call is issued until unl_cpu or iunl_cpu is issued.

## 8.5 Unlock the CPU

The CPU locked state is cancelled by issuing the following service call from the processing program.

- unl_cpu, iunl_cpu
  These service calls change the system status to the CPU unlocked state.
  As a result, acknowledge processing of maskable interrupts prohibited through issuance of either loc_cpu or iloc_cpu is enabled, and the restriction on service call issuance is released.
  If a maskable interrupt is created during the interval from when either loc_cpu or iloc_cpu is issued until this service call is issued, the RI78V4 delays transition to the relevant interrupt processing (interrupt handler) until this service call is issued.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    /* ............ */

    loc_cpu ( );                /*Lock the CPU*/

    /* ............ */          /*CPU locked state*/

    unl_cpu ( );                /*Unlock the CPU*/

    /* ............ */
}
```

Note 1 This service call does not perform queuing of cancellation requests. If the system is in the CPU unlocked state, therefore, no processing is performed but it is not handled as an error.

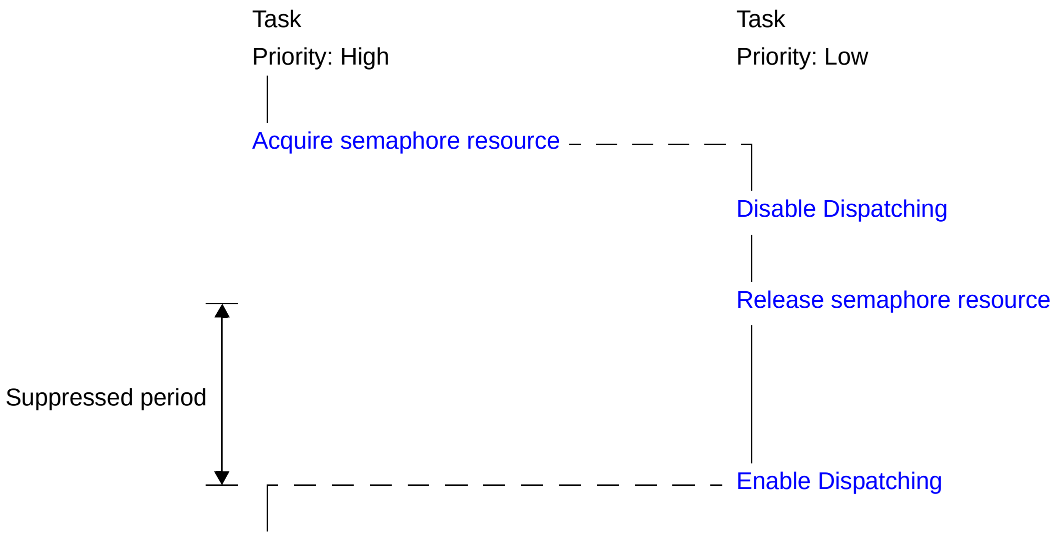
Note 2 The RI78V4 implements enabling of maskable interrupt acknowledgment bu manipulating the interrupt mask flag register (MK*xx*) and the in-service priority flag (ISP*x*) of the program status word (PSW). Therefore, manipulating of these registers from the processing program is prohibited from when loc_cpu or iloc_cpu is issued until this service call is issued.

## 8.6 Disable Dispatching

A task is moved to the dispatching disabled state by issuing the following service call from the processing program.

- dis_dsp
  This service call changes the system status to the dispatching disabled state.
  As a result, dispatch processing (task scheduling) is disabled from when this service call is issued until ena_dsp is issued.
  If a service call (chg_pri, sig_sem, etc.) accompanying dispatch processing is issued during the interval from when this service call is issued until ena_dsp is issued, the RI78V4 executes only processing such as queue manipulation, counter manipulation, etc., and the actual dispatch processing is delayed until ena_dsp is issued, upon which the actual dispatch processing is performed in batch.
  The following shows a processing flow when using this service call.

Figure 8-3 Disable Dispatching




The following describes an example for coding this service call.

```
#include        <kernel.h>      /*Standard header file definition*/
#include        <kernel_id.h>   /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    /* ............ */

    dis_dsp ( );                /*Disable dispatching*/

    /* ............ */          /*Dispatching disabled state*/

    ena_dsp ( );                /*Enable dispatching*/

    /* ............ */
}
```

Note 1 This service call does not perform queuing of disable requests. If the system is in the dispatching disabled state, therefore, no processing is performed but it is not handled as an error.

Note 2 The dispatching disabled state changed by issuing this service call must be cancelled before the task that issued this service call moves to the DORMANT state.

## 8.7 Enable Dispatching

The dispatching disabled state is cancelled by issuing the following service call from the processing program.

- ena_dsp
  This service call changes the system status to the dispatching enabled state.
  As a result, dispatch processing (task scheduling) that has been disabled by issuing dis_dsp is enabled.
  If a service call (chg_pri, sig_sem, etc.) accompanying dispatch processing is issued during the interval from when dis_dsp is issued until this service call is issued, the RI78V4 executes only processing such as queue manipulation, counter manipulation, etc., and the actual dispatch processing is delayed until this service call is issued, upon which the actual dispatch processing is performed in batch.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    /* ............ */

    dis_dsp ( );                /*Disable dispatching*/

    /* ............ */          /*Dispatching disabled state*/

    ena_dsp ( );                /*Enable dispatching*/

    /* ............ */
}
```

Note This service call does not perform queuing of enable requests. If the system is in the dispatching enabled state, therefore, no processing is performed but it is not handled as an error.

## 8.8 Reference Contexts

The context type is referenced by issuing the following service call from the processing program.

- sns_ctx
  This service call acquires the context type of the processing program that issued this service call (non-task context or task context).
  When this service call is terminated normally, the acquired context type (TRUE: non-task context, FALSE: task context) is returned.

| | |
|---|---|
| Non-task contexts: | cyclic handler, interrupt handler |
| task contexts: | task |

The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_sub ( void )
{
    BOOL    ercd;               /*Declares variable*/

    /* ............ */

    ercd = sns_ctx ( );         /*Reference contexts*/

    if ( ercd == TRUE ) {
        /* ............ */      /*Non-task contexts*/
    } else if ( ercd == FALSE ) {
        /* ............ */      /*Task contexts*/
    }

    /* ............ */
}
```

## 8.9 Reference CPU State

The CPU locked state is referenced by issuing the following service call from the processing program.

- sns_loc
  This service call acquires the system status type when this service call is issued (CPU locked state or CPU unlocked state).
  When this service call is terminated normally, the acquired system state type (TRUE: CPU locked state, FALSE: CPU unlocked state) is returned.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_sub ( void )
{
    BOOL    ercd;               /*Declares variable*/

    /* ............ */

    ercd = sns_loc ( );         /*Reference CPU state*/

    if ( ercd == TRUE ) {
        /* ............ */      /*CPU locked state*/
    } else if ( ercd == FALSE ) {
        /* ............ */      /*CPU unlocked state*/
    }

    /* ............ */
}
```

Note The system enters the CPU locked state when loc_cpu or iloc_cpu is issued, and enters the CPU unlocked state when unl_cpu or iunl_cpu is issued.

## 8.10 Reference Dispatching State

The dispatching state is referenced by issuing the following service call from the processing program.

- sns_dsp
  This service call acquires the system status type when this service call is issued (dispatching disabled state or dispatching enabled state).
  When this service call is terminated normally, the acquired system state type (TRUE: dispatching disabled state, FALSE: dispatching enabled state) is returned.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_sub ( void )
{
    BOOL    ercd;               /*Declares variable*/

    /* ............ */

    ercd = sns_dsp ( );         /*Reference dispatching state*/

    if ( ercd == TRUE ) {
        /* ............ */      /*Dispatching disabled state*/
    } else if ( ercd == FALSE ) {
        /* ............ */      /*Dispatching enabled state*/
    }

    /* ............ */
}
```

Note The system enters the dispatching disabled state when dis_dsp is issued, and enters the dispatching enabled state when ena_dsp is issued.

## 8.11 Reference Dispatch Pending State

The dispatch pending state is referenced by issuing the following service call from the processing program.

- sns_dpn
  This service call acquires the system status type when this service call is issued (whether in dispatch pending state or not).
  When this service call is terminated normally, the acquired system state type (TRUE: dispatch pending state, FALSE: dispatch not-pending state) is returned.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_sub ( void )
{
    BOOL    ercd;               /*Declares variable*/

    /* ............ */

    ercd = sns_dpn ( );         /*Reference dispatch pending state*/

    if ( ercd == TRUE ) {
        /* ............ */      /*Dispatch pending state*/
    } else if ( ercd == FALSE ) {
        /* ............ */      /*Other state*/
    }

    /* ............ */
}
```

Note The dispatch pending state designates the state in which explicit execution of dispatch processing (task scheduling processing) is prohibited by issuing either the dis_dsp, loc_cpu, or iloc_cpu service call, as well as the state during which processing of a non-task is being executed.

# CHAPTER 9 INTERRUPT MANAGEMENT FUNCTIONS

This chapter describes the interrupt management functions performed by the RI78V4.

## 9.1 Outline

The RI78V4 provides as interrupt management functions related to the interrupt handlers activated when a maskable interrupt is occurred.

In the RI78V4, interrupt servicing managed by the RI78V4 is called "interrupt handler", which is distinguished from interrupt servicing that operates without being managed by the RI78V4.

The following lists the differences between interrupt handlers and interrupt servicing.

Table 9-1 Differences Between Interrupt Handlers and Interrupt Servicing

| | Interrupt Handler | Interrupt Servicing |
|---|---|---|
| Service call issuance | Available | Not available |
| Interrupt type | Maskable interrupt | Maskable interrupt<br>Software interrupt<br>Reset interrupt |
| Interrupt priority level | Levels 2, 3 | Levels 0, 1 (*) |
| Definition in the system configuration file | Defines in DEF_INH | Not define in DEF_INH |

* It is also possible to assign a level of 2 or 3 to an application that disables multiple interrupts.

Note 1 The interrupt priority level is set using the priority specification flag register of the target CPU.

Note 2 The RI78V4 does not execute initialization of hardware that creates interrupts (clock controller, etc.). This initialization processing must therefore be coded by the user in the Boot Processing or Initialization Routine.

## 9.2 Interrupt Entry Processing

Interrupt entry processing is a routine dedicated to entry processing that is extracted as a user-own coding module to assign instructions to branch to relevant processing (such as Interrupt Handlers or Boot Processing), to the vector table address to which the CPU forcibly passes the control when an interrupt occurs.

When the interrupt handler is described by C language (the TA_HLNG attribute is specified in a interrupt handler definition of the system configuration file(DEF_INH)), the user does not have to describe interrupt entry processing because of the C compiler outputing “interrupt entry processing which corresponds to an interrupt request name“ automatically.

When the interrupt handler is described by Assembly language (the TA_ASM attribute is specified in a interrupt handler definition of the system configuration file(DEF_INH)), the user has to describe interrupt entry processing. Futher it is necessory to describe by an assembly language about branch to boot processing.

### 9.2.1 Basic form of interrupt entry processing

The code of interrupt entry processing varies depending on whether the relevant processing (Interrupt Handlers, Boot Processing, or the like) is allocated to the near area or to the far area.
The following shows examples for coding interrupt entry processing.

[ When the relevant processing (Interrupt Handlers, Boot Processing, or the like) is allocated to the near area ]

```
     .PUBLIC   _func_inthdr
_func_inthdr   .VECTOR  0x002C    ;Jump to boot processing

     .SECTION  .text, TEXT        ;Vector table address setting
_func_inthdr:

        ............              ;Main processing

  ;割り込みハンドラに制御を移す
```

[ When the relevant processing (Interrupt Handlers, Boot Processing, or the like) is allocated to the far area ]

```
     .EXTERN _intent_RESET        ;Declares symbol external reference
     .EXTERN _intent_INTTM00      ;Declares symbol external reference

     .SECTION   .vecttable, TEXT  ;Vector table section setting
_intent_RESET   .VECTOR  0x0000   ;Vector table address setting
_intent_INTTM00 .VECTOR  0x002C   ;Vector table address setting

     .SECTION  .textf, TEXTF      ;Vector table section setting
_intent_RESET:
        BR      !!_boot           ;Jump to boot processing
_intent_INTTM00:
        BR      !!_func_inthdr    ;Jump to interrupt handler
```

### 9.2.2 Internal processing of interrupt entry processing

Interrupt entry processing is a routine dedicated to processing of entries called without using the RI78V4 when an interrupt occurs. Therefore, note the following points when coding interrupt entry processing.

- Coding method
  Code interrupt entry processing in assembly language, in formats compliant with the assembler's function calling rules.

- Stack switching
  No stack requiring switching exists in interrupt entry processing execution. The code regarding stack switching during interrupt entry processing is therefore not required.

- Service call issuance
  The RI78V4 prohibits issuance of service calls in interrupt entry processing.

The following lists processing that should be executed in interrupt entry processing.

- Vector table address setting
- Passing of control to relevant processing (Interrupt Handlers, Boot Processing, or the like)

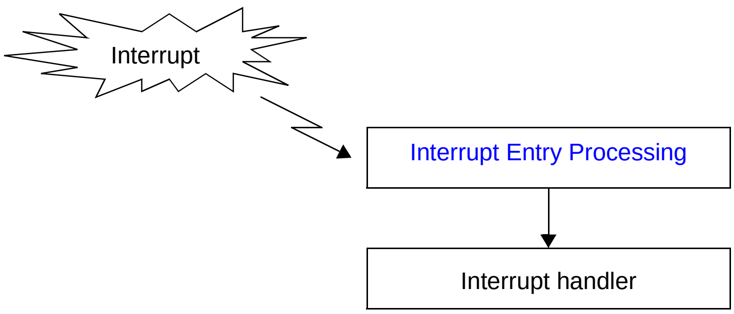
## 9.3 Interrupt Handlers

The interrupt handler is a routine dedicated to interrupt servicing that is activated when an interrupt occurs, and is called from Interrupt Entry Processing.

The RI78V4 handles the interrupt handler as a "non-task (module independent from tasks)". Therefore, even if a task with the highest priority in the system is being executed, the processing is suspended when an interrupt occurs, and the control is passed to the interrupt handler.

The following shows a processing flow from when an interrupt occurs until the control is passed to the interrupt handler.

Figure 9-1 Processing Flow (Interrupt Handler)




### 9.3.1 Define interrupt handler

Interrupt handler registration is realized by coding Interrupt Entry Processing (branch instruction to interrupt handler) to the vector table address to which the CPU forcibly passes control upon occurrence of an interrupt.

The code of Interrupt Entry Processing varies depending on whether the interrupt handler is allocated to the near area or to the far area.

When the interrupt handler is described by C language (the TA_HLNG attribute is specified in a interrupt handler definition of the system configuration file(DEF_INH)), the user does not have to describe interrupt entry processing because of the C compiler outputing “interrupt entry processing which corresponds to an interrupt request name“ automatically.

When the interrupt handler is described by Assembly language (the TA_ASM attribute is specified in a interrupt handler definition of the system configuration file(DEF_INH)), the user has to describe interrupt entry processing.

### 9.3.2 Basic form of interrupt handlers

When coding interrupt handlers in C, use void type functions that do not have arguments (any function name is fine) . The code of interrupt depending on whether the interrupt handler is allocated to the near area or to the far area. The following shows the basic form of coding interrupt handlers in C.

[ When the interrupt handler is allocated to the near area ]

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
__near func_inthdr ( void )
{
    /* ............ */          /*Main processing*/

    return;                     /*Terminate interrupt handler*/
}
```

Note 1 The TA_HLNG attribute and the TA_NEAR attribute are specified in a definition of a interrupt handler in the system configuration file (DEF_INH).

Note 2 The the #pragma rtos_interrupt directive is defined in the file “kernel_id.h” (CF78V4 outputs automatically). Therefore please the file “kernel_id.h” be sure to do include.

[ When the interrupt handler is allocated to the near area ]

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
__far func_inthdr ( void )
{
    /* ............ */          /*Main processing*/

    return;                     /*Terminate interrupt handler*/
}
```

Note 1 The TA_HLNG attribute and the TA_FAR attribute are specified in a definition of a interrupt handler in the system configuration file (DEF_INH).

Note 2 The the #pragma rtos_interrupt directive is defined in the file “kernel_id.h” (CF78V4 outputs automatically). Therefore please the file “kernel_id.h” be sure to do include.

When coding interrupt handlers in assembly language, use void type functions that do not have arguments (function: any).

The code of interrupt depending on whether the interrupt handler is allocated to the near area or to the far area.

The following shows the basic form of coding interrupt handlers in assembly language.

- When the interrupt handler is allocated to the near area
  Saves AX register and stores the vector table address when an interrupt occurs, calls processing to switch to the system stack (function name: _kernel_int_entry), and then call end processing at the end of the interrupt handler (function name: _kernel_int_exit). A near attribute section is specified as an allocated section of the interrupt handler.

[ When the interrupt handler is allocated to the near area in assembly language ]

```
$INCLUDE        (kernel.inc)     ;Standard header file definition
$INCLUDE        (kernel_id.inc)  ;System information header file definition
     .PUBLIC   _func_inthdr

_func_inthdr   .VECTOR  0x002C   ;Switches to system stack, Saves registers

     .SECTION  .text, TEXT
_func_inthdr:                    ;Interrupt handler

     PUSH   AX                   ;Saves AX register
     MOV    AX, #0x002C          ;Stores the vector table address when an interrupt
                                 ;occurs

     CALL   !!__kernel_int_entry ;Switchs to the system stack and saves registers

        ............             ;Main processing

     BR     !!__kernel_int_exit  ;Terminate interrupt handler, Restores registers
```

Note The TA_ASM attribute and the TA_NEAR attribute are specified in a definition of a interrupt handler in the system configuration file (DEF_INH).

- When the interrupt handler is allocated to the far area
  Calls processing to switch to the system stack (function name: _kernel_int_entry), and then call end processing at the end of the interrupt handler (function name: _kernel_int_exit). A far attribute section is specified as an allocated section of the interrupt handler.

```
$INCLUDE        (kernel.inc)     ;Standard header file definition
$INCLUDE        (kernel_id.inc)  ;System information header file definition
     .PUBLIC   _func_inthdr

_func_inthdr   .VECTOR  0x002C

     .SECTION  .textf, TEXTF    ;Interrupt handler
_func_inthdr:
     CALL   !!__kernel_int_entry ;Switchs to the system stack and saves registers

        ............             ;Main processing

     BR     !!__kernel_int_exit  ;Terminate interrupt handler, Restores registers
```

Note 1 The TA_ASM attribute and the TA_FAR attribute are specified in a definition of a interrupt handler in the system configuration file (DEF_INH).

Note 2 When allocating a interrupt handler in far area, the far branch information is needed. In other words, it jumps from the vector table address to far branch information and jumps to a interrupt handler from there. CF78V4 outputs this far branch information automatically in a system information table file.

Note 3 CF78V4 outputs processing code that saves AX register and sets the vector table address of the factor automatically in a system information table file.

### 9.3.3 Internal processing of interrupt handler

The RI78V4 handles the interrupt handler as a "non-task".

Moreover, the RI78V4 executes "original pre-processing" when passing control to the interrupt handler, as well as "original post-processing" when regaining control from the interrupt handler.

Therefore, note the following points when coding interrupt handlers.

- Coding method
  Code interrupt handlers using C or assembly language in the format shown in "9.3.2 Basic form of interrupt handlers".

- Stack switching
  When the interrupt handler is described by C language, the user does not have to describe to switch to the system stack (calls _kernel_int_entry) because of the C compiler outputing this code automatically. When the interrupt handler is described by assembly language, saves AX register and stores the vector table address when an interrupt occurs, calls processing to switch to the system stack (function name: _kernel_int_entry), and then call end processing at the end of the interrupt handler (function name: _kernel_int_exit).

- Saving/storing of data in register
  When the interrupt handler is described by C language, the user does not have to describe to switch to the system stack (calls _kernel_int_entry) because of the C compiler outputing this code automatically. When the interrupt handler is described by assembly language, data of general-purpose registers (AX, BC, DE, HL) and registers ES CS is saved and restored in that function execution, by explicitly calling register data save processing (function name: _kernel_int_entry) at the beginning of the interrupt handler, and calling data restore processing (function name: _kernel_int_exit) at the end of the interrupt handler.

  Note 1 Data of the PSW and PC are automatically saved and stored by the CPU.

- Interrupt status
  The RI78V4 goes into the following state when passing control to an interrupt handler.
  Consequently, after control has passed to an interrupt handler, if an interrupt occurs with a higher precedence than the current level, then multiple interrupts can be processed.

  - Acceptance of maskable interrupts is permitted
    IE = 1

  - Interrupts with the precedence below are disabled
    A level-2 interrupt handler process is ongoing: ISP1 = 0, ISP0 = 1
    A level-3 interrupt handler process is ongoing: ISP1 = 1, ISP0 = 0

    Note It is not possible to define level 0 or 1 as an interrupt handler.

  Note Even if the acceptance of maskable interrupts is disabled inside an interrupt handler (IE = 0), it will be enabled (IE = 1) after control returns from the interrupt handler.

- Service call issuance
  The RI78V4 handles the interrupt handler as a "non-task".
  Service calls that can be issued in interrupt handlers are limited to the service calls that can be issued from non-tasks.

  Note 1 For details on the valid issuance range of each service call, refer to Table 12-8 to Table 12-17.

  Note 2 If a service call (ichg_pri, isig_sem, etc.) accompanying dispatch processing (task scheduling processing) is issued in order to quickly complete the processing in the interrupt handler during the interval until the processing in the interrupt handler ends, the RI78V4 executes only processing such as queue manipulation, counter manipulation, etc., and the actual dispatch processing is delayed until a return instruction is issued by the interrupt handler, upon which the actual dispatch processing is performed in batch.

## 9.4 Controlling Enabling/Disabling of Interrupts

### 9.4.1 Interrupt level under management of the RI78V4

The microcontroller manages four levels of interrupts: level 0 to level 3. On the RI78V4, the interrupt levels at which service calls can be issued from an interrupt are permanently set to levels 2 and 3, these are treated as the interrupt levels managed by the RI78V4.

- Interrupt levels 2 and 3 are managed by the RI78V4.
  Service calls can be issued from levels 2 and 3. Interrupt handlers, which are interrupts (including timer interrupts) managed by the RI78V4, must be set to level 2 or 3.

- Interrupt levels 0 and 1 are not managed by the RI78V4
  Service calls cannot be issued from levels 0 or 1. Behavior is not guaranteed if a service call is issued from level 0 or 1. Interrupt processes, which are interrupts not managed by the RI78V4, must be set to level 0 or 1. There is, however, an exception: user applications that disable multiple interrupts (see below) can set interrupts to level 2 or 3.

### 9.4.2 Controlling enabling/disabling of interrupts in the RI78V4

The RI78V4 uses the "ISP1" and "ISP0" bits in the PSW register to enable and disable interrupts. Set ISP1 to 0 and ISP0 to 1 to disable interrupts in the RI78V4. Set ISP1 to 1 and ISP0 to 1 to enable interrupts in the RI78V4.

Figure 9-2 ISP1 and ISP0 Bits in PSW Register

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| PSW register | IE | Z | RBSB1 | AC | RBSB0 | ISP1 | ISP0 | CY |

| ISP1 | ISP0 | Interrupt priority in current process |
|---|---|---|
| 0 | 0 | Enable level-0 interrupts<br>(While processing a level-1 or level-0 interrupt) |
| 0 | 1 | Enable level-0 and level-1 interrupts<br>(While processing a level-2 interrupt) |
| 1 | 0 | Enable level-0, level-1, and level-2 interrupts<br>(While processing a level-3 interrupt) |
| 1 | 1 | Enable all interrupts<br>(Standby for the acceptance of interrupts) |

The "IE" bit of the RI78V4's PSW register inherits the value of the service call or RI78V4-function issuer. EI and DI instructions do not manipulate the "IE" value. As exceptions, however, there are places in the RI78V4 where EI and DI instructions are used.

- Immediately before starting a task specifying interrupts as disabled, a DI instruction is used to set IE to 0.
- Immediately before starting a task specifying interrupts as enabled, an EI instruction is used to set IE to 1.
- Immediately before starting the idle routine, an EI instruction is used to set IE to 1.
- Inside the __kernel_int_entry function, which performs interrupt handler start processing, IE is set to 1.

### 9.4.3 Controlling enabling/disabling of interrupts in user processes

User applications use the EI function (or EI instruction) and DI function (or DI instruction) to manipulate interrupts. In a task or other user process, using the DI function disables all maskable interrupts from being accepted; using the EI function enables maskable interrupts to be accepted in accordance with the state of the "ISP1" and "ISP0" bits.

The RI78V4 sets whether interrupts are enabled or disabled upon start of the user process. The states are listed below.

Table 9-2 States Enabling and Disabling Interrupts upon Process Start

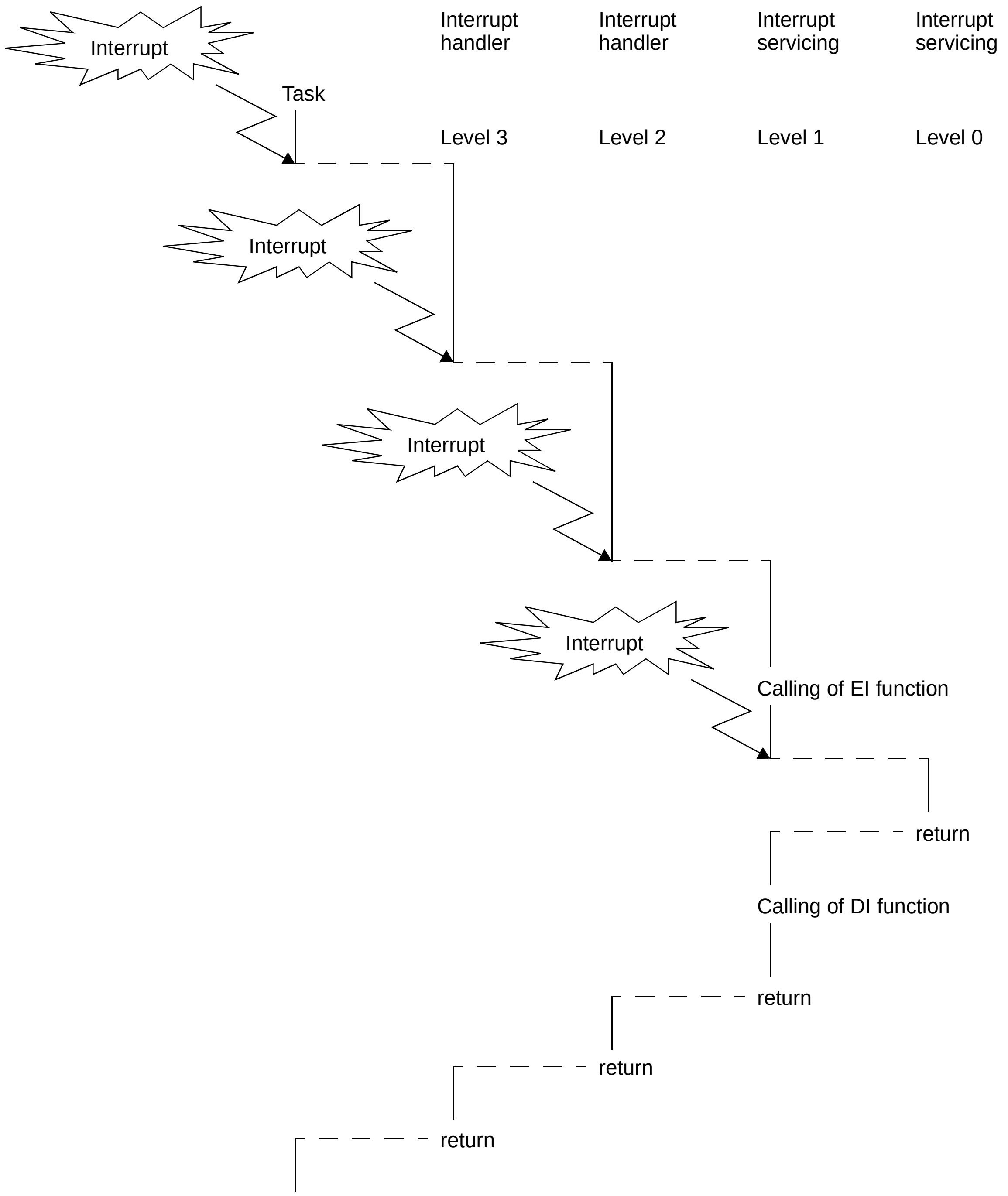
| Process to Start | | IE | ISP1 | ISP0 | Interrupt Enabled/Disabled on Start |
|---|---|---|---|---|---|
| Initialization routine | | 0 | 1 | 1 | Interrupts disabled (behavior is not guaranteed when it is enabled by the process) |
| Idle routine | | 1 | 1 | 1 | Interrupts enabled; all interrupt levels accepted |
| Task | When interrupts specified as enabled | 1 | 1 | 1 | Interrupts enabled; all interrupt levels accepted |
| | When interrupts specified as disabled | 0 | 1 | 1 | Interrupts disabled (if enabled, all interrupt levels accepted) |
| Cyclic handler | When a level-2 interrupt occurs | 1 | 0 | 1 | Interrupts enabled; level-0 and level-1 levels accepted |
| | When a level-3 interrupt occurs | 1 | 1 | 0 | Interrupts enabled; level-0, level-1, and level-2 levels accepted |
| Interrupt handler | When a level-2 interrupt occurs | 1 | 0 | 1 | Interrupts enabled; level-0 and level-1 levels accepted |
| | When a level-3 interrupt occurs | 1 | 1 | 0 | Interrupts enabled; level-0, level-1, and level-2 levels accepted |
| Interrupt servicing | When a level-0 interrupt occurs | 0 | 0 | 0 | Interrupts disabled (if enabled, a lelvel-0 interrupt accepted) |
| | When a level-1 interrupt occurs | 0 | 0 | 0 | Interrupts disabled (if enabled, a lelvel-0 interrupt accepted) |
| | When a level-2 interrupt occurs | 0 | 0 | 1 | Interrupts disabled (if enabled, lelvel-0 and lebel-1 interrupts accepted) |
| | When a level-3 interrupt occurs | 0 | 1 | 0 | Interrupts disabled (if enabled, lelvel-0, level-1, and lebel-2 interrupts accepted) |

Note that a separate "IE" state is maintained for each task. If a suspended task is resumed, the IE state before suspension is restored.

## 9.5 Multiple Interrupts

The reoccurrence of an interrupt within an interrupt handler is called "multiple interrupt".
The following shows the flow of the processing for handling multiple interrupts.

Figure 9-3 Multiple Interrupts



When control moves to an interrupt handler, then the state changes to acceptance of maskable interrupts enabled ("IE = 1"). For this reason, multiple interrupts are generally accepted from interrupt handlers. Multiple interrupts are likewise accepted from timer interrupts and cyclic handlers called from them.

When control moves to an interrupt process, then the state changes to acceptance of maskable interrupts disabled (because the RI78V4 does not mediate, the behavior is in accordance with that of the microcontroller). For this reason, multiple interrupts are generally not accepted from interrupt processes. To enable the acceptance of multiple interrupts, it is necessary to call the EI function from the interrupt process. It is not allowed to accept multiple interrupt handlers from an interrupt process, and behavior is not guaranteed if this occurs.

If a user application enables multiple interrupts, then it is necessary to set the interrupt level of the interrupt handler/process as shown below.

Table 9-3 Settable Interrupt Level (Enabling Multiple Interrupts from User Application)

| | Interrupt Handler | Interrupt Servicing |
|---|---|---|
| Interrupt level 0 | Not available | Available |
| Interrupt level 1 | Not available | Available |
| Interrupt level 2 | Available | Not available |
| Interrupt level 3 | Available | Not available |

If a user application disables multiple interrupts, then it is necessary to set the interrupt level of the interrupt handler/process to one of the patterns shown below.

Pattern 1: Set the level of all interrupt handlers and interrupt processes to 2.
Pattern 2: Set the level of all interrupt handlers and interrupt processes to 3.
Pattern 3: Set the level of all interrupt handlers and to 2, and the level of all interrupt processes to either 2 or 3. Interrupts are disabled during an interrupt process with an interrupt level of 3 (IE = 0).

Table 9-4 Settable Interrupt Level (Disabling Multiple Interrupts from User Application)

| | Pattern 1 | | Pattern 2 | | Pattern 3 | |
|---|---|---|---|---|---|---|
| | Interrupt Handler | Interrupt Servicing | Interrupt Handler | Interrupt Servicing | Interrupt Handler | Interrupt Servicing |
| Interrupt level 0 | Not available | Not available | Not available | Not available | Not available | Not available |
| Interrupt level 1 | Not available | Not available | Not available | Not available | Not available | Not available |
| Interrupt level 2 | Available | Available | Not available | Not available | Available | Available |
| Interrupt level 3 | Not available | Not available | Available | Available | Not available | Available (*) |

(*) Interrupts are disabled during this interrupt process (IE = 0).

# CHAPTER 10 SYSTEM CONFIGURATION MANAGEMENT FUNCTIONS

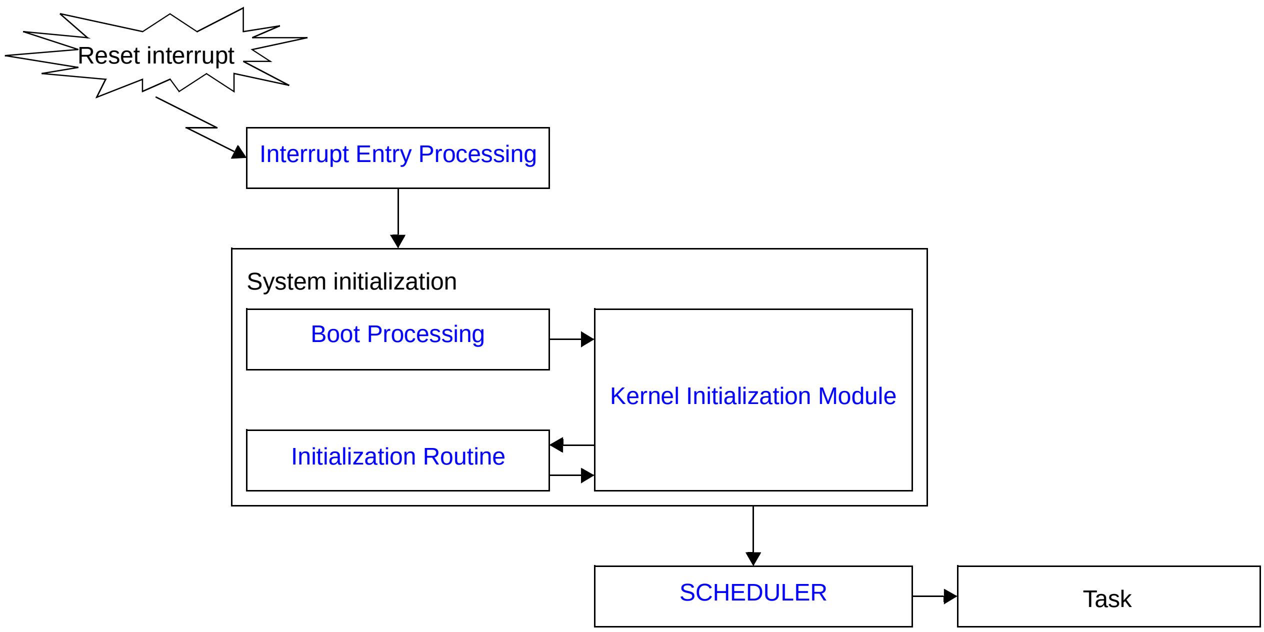
This chapter describes the system configuration management functions performed by the RI78V4.

## 10.1 Outline

The system configuration management functions of the RI78V4 provides system initialization processing, which is required from the reset interrupt output until control is passed to the task, and version information referencing processing.

The following shows a processing flow from when a reset interrupt occurs until the control is passed to the task.

Figure 10-1 Processing Flow (System Initialization)



## 10.2 Boot Processing

Boot processing is a routine dedicated to initialization processing that is extracted as a user-own coding module to initialize the minimum required hardware for the RI78V4 to perform processing. Boot processing is called from Interrupt Entry Processing that is assigned to the vector table address to which the CPU forcibly passes the control when a reset interrupt occurs.

### 10.2.1 Define boot processing

Boot processing registration is realized by coding Interrupt Entry Processing (branch instruction to boot processing) to the vector table address to which the CPU forcibly passes control upon occurrence of a reset interrupt.

The code of Interrupt Entry Processing varies depending on whether boot processing is allocated to the near area or to the far area.

The following shows examples for coding Interrupt Entry Processing.

[ When boot processing is allocated to the near area ]

```
      .PUBLIC    _boot       ;Vector table address setting
_boot    .VECTOR  0x0000     ;Jump to boot processing _boot
```

[ When boot processing is allocated to the far area ]

```
      .EXTERN    _intent_RESET     ;Declares symbol external reference

      .SECTION   .vecttable, TEXT  ;Vector table section setting
_intent_RESET   .VECTOR  0x0000    ;Vector table address setting

      .SECTION  .textf, TEXTF      ;Vector table address setting
_intent_RESET:
         BR      !!_boot           ;Jump to boot processing _boot
```

### 10.2.2 Basic form of boot processing

Write Boot processing as a function that does not include arguments and return values (function name: any name).

The following shows the basic form of boot processing.

```
        .PUBLIC   _boot
        .EXTERN   __kernel_start, _hdwinit, __init_ri_stackarea, _reset

        .SECTION .stack_bss, BSS        ;Sets stack section
_stackend:
        .DS       0x100
_stacktop:

_boot   .VECTOR  0x0000

        .SECTION .text, TEXT
_boot:
        SEL      RB0                    ;Sets register bank

        MOVW     SP, #LOWW(_stacktop)   ;Sets stack pointer SP

        CALL     !!_reset

;Clears initial information items of RI78V4
        MOVW     HL, #LOWW(STARTOF(.kernel_data_init))
        MOVW     AX, #LOWW(STARTOF(.kernel_data_init) + SIZEOF(.kernel_data_init))
        BR       $L2_KERNEL_DATA
L1_KERNEL_DATA:
        MOV      [HL+0], #0
        INCW     HL
L2_KERNEL_DATA:
        CMPW     AX, HL
        BNZ      $L1_KERNEL_DATA

        CALL     !!__init_ri_stackarea  ;Clears RAM area

        BR       !!__kernel_start       ;Jump to Kernel Initialization Module

        CLRW     AX
_exit:
        BR       $exit
```

### 10.2.3 Internal processing of boot processing

Boot processing is a routine dedicated to initialization processing called from Interrupt Entry Processing without using the RI78V4. Therefore, note the following points when coding boot processing.

- Coding method
  Code boot processing in assembly language.

- Stack switching
  Setting of stack pointer SP is not executed at the point when control is passed to boot processing.
  To use a boot processing dedicated stack, setting of stack pointer SP must therefore be coded at the beginning of the boot processing.

- Interrupt status
  The Kernel Initialization Module is not executed at the point when control is passed to boot processing. The system may therefore hang up when an interrupt is created before the processing is completed. To avoid this, explicitly prohibit acknowledgment of maskable interrupts by manipulating interrupt enable flag IE of program status word PSW during boot processing.

- Register bank setting
  The RI78V4 prohibits switching of a register bank that was set before __urx_start is called in boot processing to another register bank (except for the case when interrupt servicing not managed by the RI78V4).

- Service call issuance
  The RI78V4 prohibits issuance of service calls in boot processing.

The following lists processing that should be executed in boot processing.

- Setting of stack pointer SP
- Setting of interrupt enable flag IE
- Initialization of internal units and peripheral controllers
- Initialization of RAM area (initialization of memory area without initial value, copying of initialization data)
- Passing of control to Kernel Initialization Module (function name: _urx_start)

Note Setting of stack pointer SP is required only when a stack dedicated to boot processing is used in boot processing.

### 10.2.4 System dependence information

System dependence information is the header file as the user own cording part which need for RI78V4 processing (file name : usrown.h).

- Basic form of system dependence information
  When describes system dependence information, uses the prescribed file name (usrown.h), the prescribed macro name (KERNEL_USR_TMCNTREG, KERNEL_USR_TMCMPREG).
  The following shows the basic form of system dependent information using C language

```
#include    <kernel_id.h>          /*System information header file definition*/

#define     KERNEL_USR_TMCNTREG 0x0180      /*I/O address */
#define     KERNEL_USR_TMCMPREG 0xff18      /*I/O address */
```

The following shows the list of the information which should be defined as system dependence information.

- Definition of system information header file
  The inclusion of system information header file output by CF78V4

  Note Only the case selected “Taking in long statistics by software trace mode” is needed description (Property panel -> [Task Analyzer] tab -> [Trace] -> [Selection of trace mode])

- Information of the clock timer
  Macro definition of the I/O address of the clock timer and the I/O address of the compare register.

  Note Only the case selected “Taking in long statistics by software trace mode” is needed description (Property panel -> [Task Analyzer] tab -> [Trace] -> [Selection of trace mode])

## 10.3 Initialization Routine

The initialization routine is a routine dedicated to initialization processing that is extracted as a user-own coding module to initialize the hardware dependent on the user execution environment (such as the peripheral controller), and is called from the Kernel Initialization Module.

### 10.3.1 Define initialization routine

In the RI78V4, the method of registering an initialization routine is limited to "static registration by the Kernel Initialization Module".

Initialization routines therefore cannot be created dynamically using a method such as issuing a service call from a processing program.

- Static define
  Static initialization routine registration is realized by coding initialization routines by using the prescribed function name init_handler.
  The RI78V4 executes initialization routine registration processing based on relevant symbol information, using the Kernel Initialization Module, and handles the registered initialization routines as management targets.

### 10.3.2 Undefine initialization routine

In the RI78V4, initialization routines registered statically by the Kernel Initialization Module cannot be unregistered dynamically using a method such as issuing a service call from a processing program.

### 10.3.3 Basic form of initialization routine

Write initialization routines using void type functions that do not have arguments (function: init_handler).
The following shows the basic form of initialization routine.

[ C Language ]

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
init_handler ( void )
{
    /* ............ */          /*Main processing*/

    return;                     /*Terminate initialization routine*/
}
```

[ Assembly Language ]

```
$INCLUDE   (kernel.inc)            ;Standard header file definition
$INCLUDE   (kernel_id.inc)         ;System information header file definition

        .PUBLIC _init_handler

        .SECTION .textf, TEXTF
_init_handler:
        ............               ;Main processing
        ............

        RET                        ;Terminate initialization routine
```

### 10.3.4 Internal processing of initialization routine

Moreover, the RI78V4 executes "original pre-processing" when passing control to the initialization routine, as well as "original post-processing" when regaining control from the initialization routine.
Therefore, note the following points when coding initialization routines.

- Coding method
  Code initialization routines using C or assembly language in the format shown in "10.3.3 Basic form of initialization routine".

- Stack switching
  The RI78V4 executes processing to switch to the system stack when passing control to the initialization routine, and processing to switch to the stack for the Kernel Initialization Module when regaining control from the initialization routine.
  The user is therefore not required to code processing related to stack switching in initialization routines.

- Interrupt status
  Maskable interrupt acknowledgement is prohibited in the RI78V4 when control is passed to the initialization routine.
  Kernel Initialization Module is not completed at the point when control is passed to the initialization routine. The system may therefore hang up when acknowledgment of maskable interrupts is explicitly enabled within the initialization routine. Therefore, enabling maskable interrupt acknowledgment in the initialization routine is prohibited in the RI78V4.

- Service call issuance
  The RI78V4 prohibits issuance of service calls in initialization routines.

The following lists processing that should be executed in initialization routines.

- Initialization of internal units and peripheral controllers
- Initialization of RAM area (initialization of memory area without initial value, copying of initialization data)
- Returning of control to Kernel Initialization Module

## 10.4 Kernel Initialization Module

The kernel initialization module is a dedicated initialization processing routine provided for initializing the minimum required software for the RI78V4 to perform processing, and is called from Boot Processing.
The following processing is executed in the kernel initialization module.

- Securement of memory area
- Creating and registering management objects
- Calling of initialization routine
- Passing of control to scheduler

Note The kernel initialization module is part of the functions provided by the RI78V4. The user therefore need not code the processing contents of the kernel initialization module.

## 10.5 Reference Version Information

Version information is referenced by issuing the following service call from the processing program.

- ref_ver
  The service call stores version information packet (such as kernel maker's code) to the area specified by parameter *pk_rver*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    T_RVER  pk_rver;            /*Declares data structure*/
    UH      maker;              /*Declares variable*/
    UH      prid;               /*Declares variable*/
    UH      spver;              /*Declares variable*/
    UH      prver;              /*Declares variable*/
    UH      prno[4];            /*Declares variable*/

    /* ............ */

    ref_ver ( &pk_rver );       /*Reference version information*/

    maker = pk_rver.maker;      /*Reference Kernel maker's code*/
    prid = pk_rver.prid;        /*Reference identification number of the kernel*/
    spver = pk_rver.spver;      /*Reference version number of the ITRON
                                  Specification*/
    prver = pk_rver.prver;      /*Reference version number of the kernel*/
    prno[0] = pk_rver.prno[0];  /*Reference management information of the kernel
                                  product (version type)*/
    prno[1] = pk_rver.prno[1];  /*Reference management information of the kernel
                                  product (memory model)*/

    /* ............ */
}
```

Note For details about the version information packet, refer to "12.5.9 Version information packet".

# CHAPTER 11 SCHEDULER

This chapter describes the scheduler of the RI78V4.

## 11.1 Outline

The scheduling functions provided by the RI78V4 consist of functions manage/decide the order in which tasks are executed by monitoring the transition states of dynamically changing tasks, so that the CPU use right is given to the optimum task.

## 11.2 Driving Method

The RI78V4 employs the Event-driven system in which the scheduler is activated when an event (trigger) occurs.

- Event-driven system

  Under the event-driven system of the RI78V4, the scheduler is activated upon occurrence of the events listed below and dispatch processing (task scheduling processing) is executed.

  - Issuance of service call that may cause task state transition
  - Issuance of instruction for returning from non-task (cyclic handler, interrupt handler, etc.)
  - Occurrence of clock interrupt used when achieving TIME MANAGEMENT FUNCTIONS

## 11.3 Scheduling System

As task scheduling methods, the RI78V4 employs the Priority level method, which uses the priority level defined for each task, and the FCFS method, which uses the time elapsed from the point when a task becomes subject to the RI78V4 scheduling.

- Priority level method

  A task with the highest priority level is selected from among all the tasks that have entered an executable state (RUNNING state or READY state), and given the CPU use right.

  Note In the RI78V4, a task having a smaller priority number is given a higher priority.

- FCFS method

  The same priority level can be defined for multiple tasks in the RI78V4. Therefore, multiple tasks with the highest priority level, which is used as the criterion for task selection under the Priority level method, may exist simultaneously.
  To remedy this, dispatch processing (task scheduling processing) is executed on a first come first served (FCFS) basis, and the task for which the longest interval of time has elapsed since it entered an executable state (READY state) is selected as the task to which the CPU use right is granted.

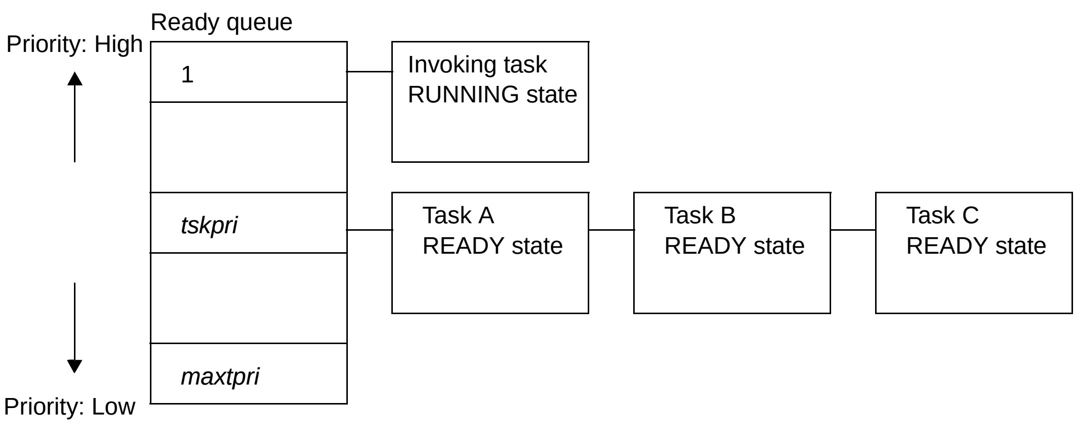
## 11.4 Ready Queue

The RI78V4 uses a "ready queue" to implement task scheduling.

The ready queue is a hash table that uses priority as the key, and tasks that have entered an executable state (READY state or RUNNING state) are queued in FIFO order. Therefore, the scheduler realizes the RI78V4's scheduling method (priority level or FCFS) by executing task detection processing from the highest priority level of the ready queue upon activation, and upon detection of queued tasks, giving the CPU use right to the first task of the proper priority level.

The following shows the case where multiple tasks are queued to a ready queue.

Figure 11-1 Implementation of Scheduling Method (Priority Level Method or FCFS Method)




### 11.4.1 Create ready queue

In the RI78V4, the method of creating a ready queue is limited to "static creation by the Kernel Initialization Module".

Ready queues therefore cannot be created dynamically using a method such as issuing a service call from a processing program.

- Static create
  Static ready queue creation is realized by defining Task priority information in the system configuration file.
  The RI78V4 executes ready queue creation processing based on data stored in information files, using the Kernel Initialization Module, and handles the created ready queues as management targets.

### 11.4.2 Delete ready queue

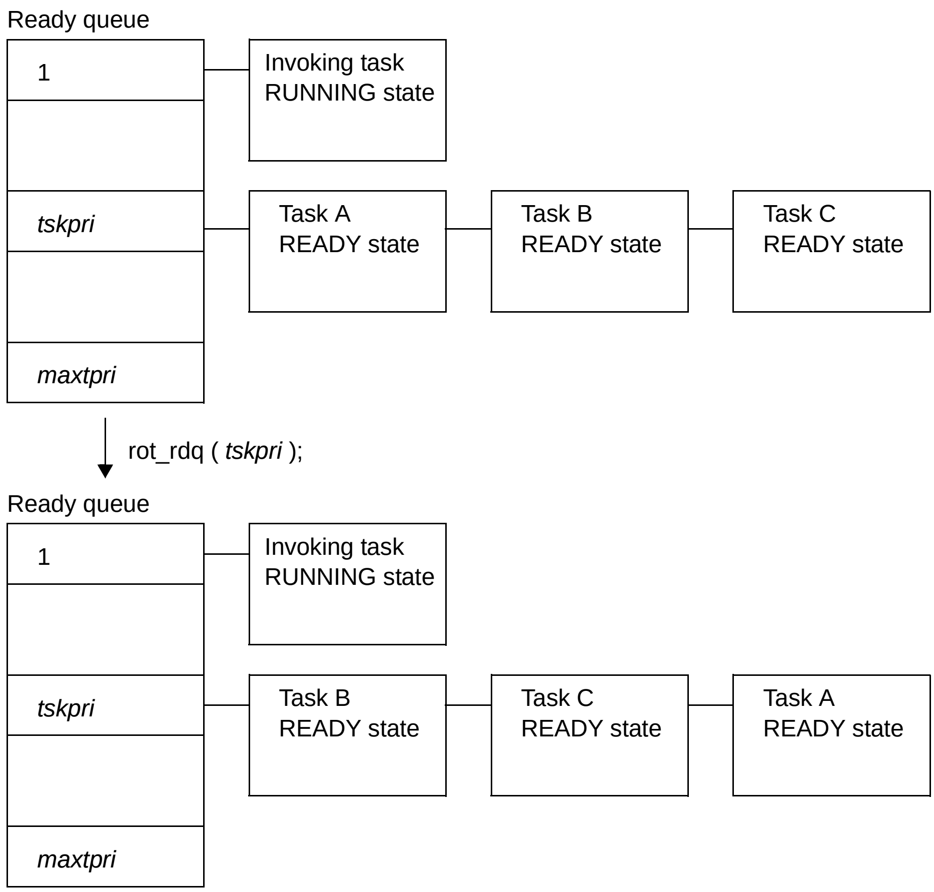
In the RI78V4, ready queues created statically by the Kernel Initialization Module cannot be deleted dynamically using a method such as issuing a service call from a processing program.

### 11.4.3 Rotate task precedence

The RI78V4 provides a function to change the queuing order of tasks from the processing program, explicitly switching the task execution order.

The following shows the status transition when the task queuing order is changed.

Figure 11-2 Rotate Task Precedence



A ready queue is rotated by issuing the following service call from the processing program.

- rot_rdq, irot_rdq
  These service calls re-queue the first task of the ready queue corresponding to the priority specified by parameter *tskpri* to the end of the queue to change the task execution order explicitly.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_cychdr ( void )
{
    PRI     tskpri = 8;         /*Declares and initializes variable*/

    /* ............ */

    irot_rdq ( tskpri );        /*Rotate task precedence*/

    /* ............ */

    return;                     /*Terminate cyclic handler*/
}
```

Note 1 This service call does not perform queuing of rotation requests. If no task is queued to the ready queue corresponding to the relevant priority, therefore, no processing is performed but it is not handled as an error.

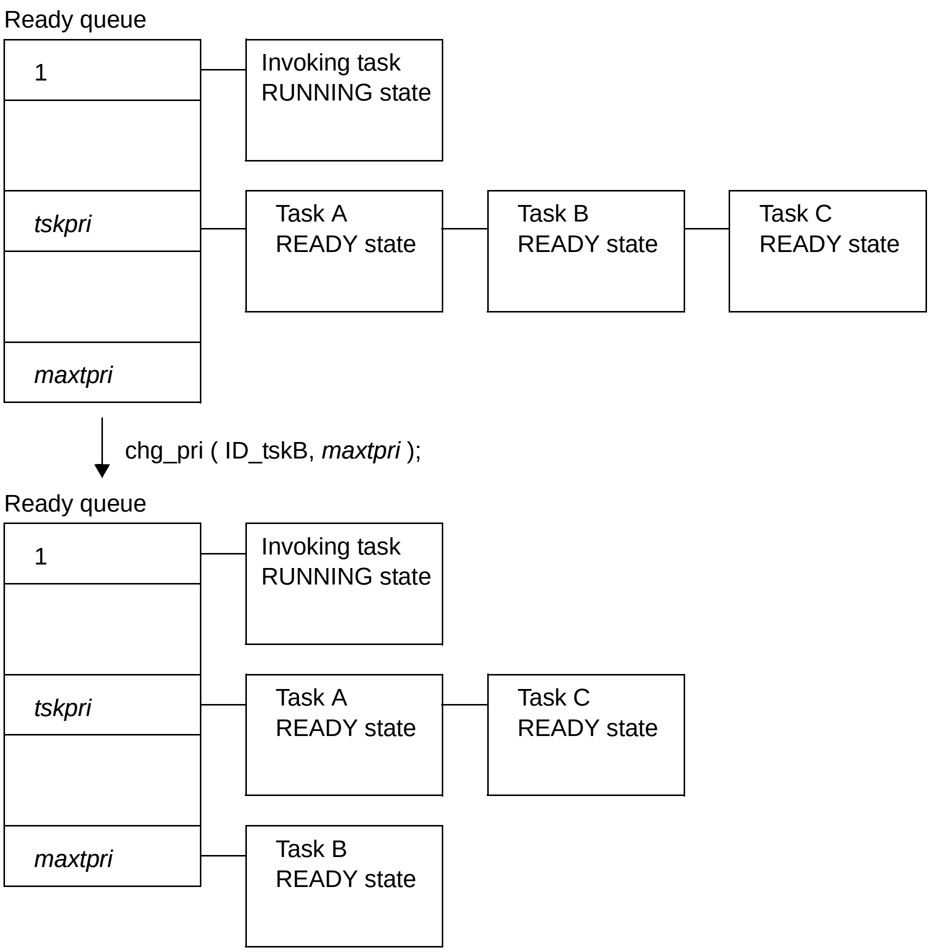
Note 2 Round-robin scheduling can be implemented by issuing this service call via a cyclic handler in a constant cycle.

### 11.4.4 Change task priority

The RI78V4 provides a function to change the priority level of tasks from the processing program, explicitly switching the task execution order.

The following shows the status transition when this task priority is changed.

Figure 11-3 Change Task Priority



A priority is changed by issuing the following service call from the processing program.

- chg_pri, ichg_pri
  This service call changes the priority of the task specified by parameter *tskid* (current priority) to a value specified by parameter *tskpri*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ID      tskid = ID_tskA;    /*Declares and initializes variable*/
    PRI     tskpri = 9;         /*Declares and initializes variable*/

    /* ............ */

    chg_pri ( tskid, tskpri );  /*Change task priority*/

    /* ............ */
}
```

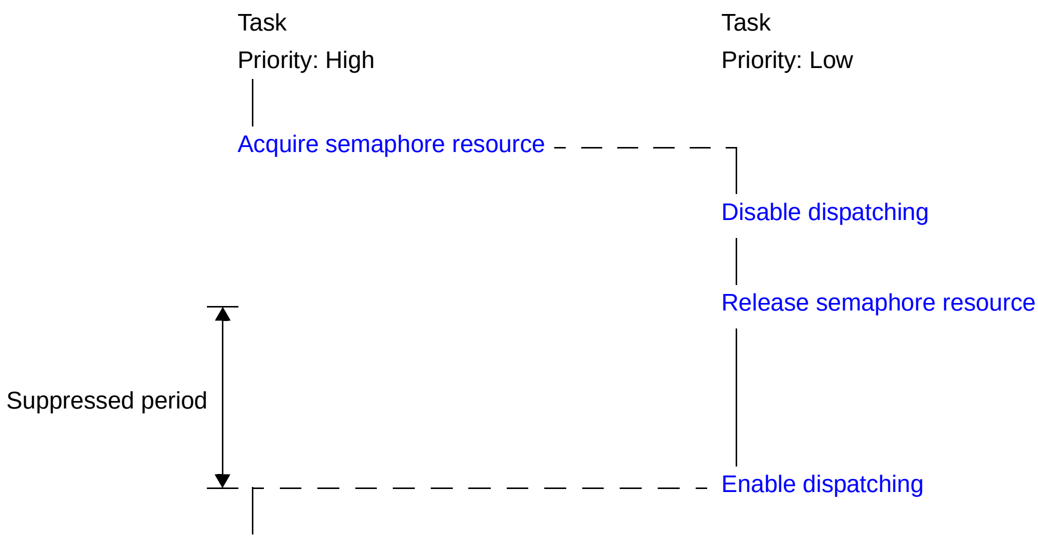
Note If the target task is in the RUNNING or READY state after this service call is issued, this service call re-queues the task at the end of the ready queue corresponding to the priority specified by parameter *tskpri*, following priority change processing.

## 11.5 Scheduling Disabling

The RI78V4 provides a function to disable scheduler activation by referencing the system state from the processing program and explicitly prohibiting dispatch processing (task scheduling processing).

The following shows a processing flow when using the scheduling suppressing function.

Figure 11-4 Scheduling Suppression Function



### 11.5.1 Disable dispatching

A task is moved to the dispatching disabled state by issuing the following service call from the processing program.

- dis_dsp
  This service call changes the system status to the dispatching disabled state.
  As a result, dispatch processing (task scheduling) is disabled from when this service call is issued until ena_dsp is issued.
  If a service call (chg_pri, sig_sem, etc.) accompanying dispatch processing is issued during the interval from when this service call is issued until ena_dsp is issued, the RI78V4 executes only processing such as queue manipulation, counter manipulation, etc., and the actual dispatch processing is delayed until ena_dsp is issued, upon which the actual dispatch processing is performed in batch.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    /* ............ */

    dis_dsp ( );                /*Disable dispatching*/

    /* ............ */          /*Dispatching disabled state*/

    ena_dsp ( );                /*Enable dispatching*/

    /* ............ */
}
```

Note 1 This service call does not perform queuing of disable requests. If the system is in the dispatching disabled state, therefore, no processing is performed but it is not handled as an error.

Note 2 The dispatching disabled state changed by issuing this service call must be cancelled before the task that issued this service call moves to the DORMANT state.

### 11.5.2 Enable dispatching

The dispatching disabled state is cancelled by issuing the following service call from the processing program.

- ena_dsp
  This service call changes the system status to the dispatching enabled state.
  As a result, dispatch processing (task scheduling) that has been disabled by issuing dis_dsp is enabled.
  If a service call (chg_pri, sig_sem, etc.) accompanying dispatch processing is issued during the interval from when dis_dsp is issued until this service call is issued, the RI78V4 executes only processing such as queue manipulation, counter manipulation, etc., and the actual dispatch processing is delayed until this service call is issued, upon which the actual dispatch processing is performed in batch.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    /* ............ */

    dis_dsp ( );                /*Disable dispatching*/

    /* ............ */          /*Dispatching disabled state*/

    ena_dsp ( );                /*Enable dispatching*/

    /* ............ */
}
```

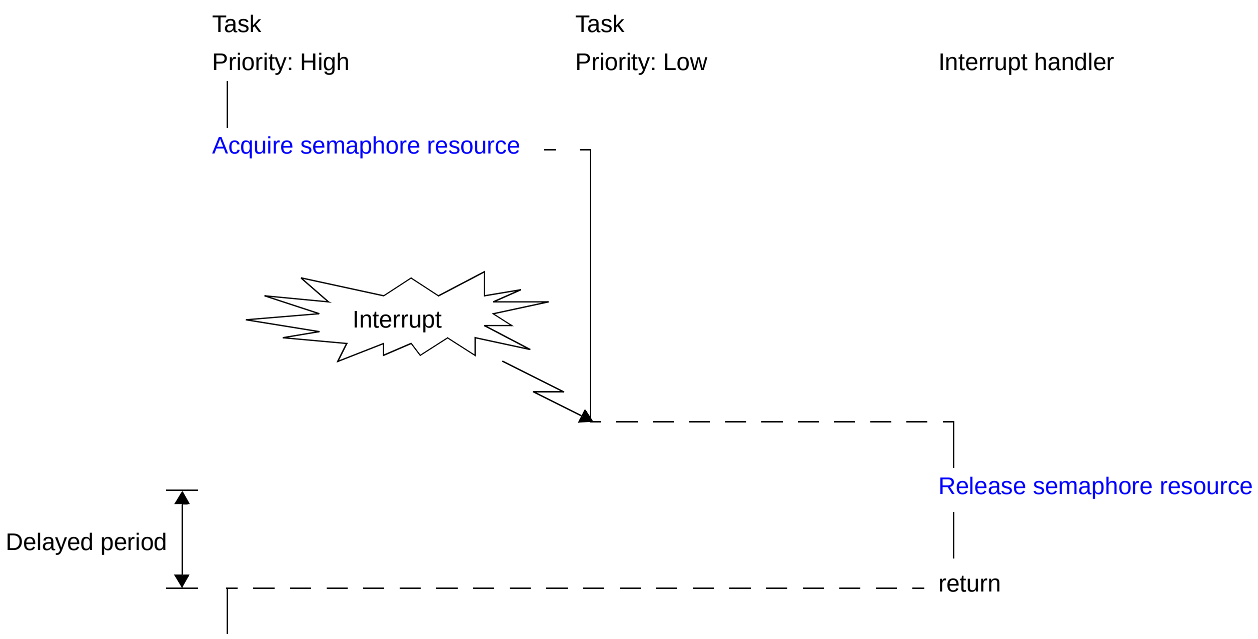
Note This service call does not queue enable requests. If the system is in the dispatching enabled state, therefore, no processing is performed but it is not handled as an error.

## 11.6 Delay of Scheduling

If a service call (ichg_pri, isig_sem, etc.) accompanying dispatch processing (task scheduling processing) is issued in order to quickly complete the processing in a non-task (cyclic handler, interrupt handler, etc.) during the interval until the processing in the non-task ends, the RI78V4 executes only processing such as queue manipulation, counter manipulation, etc., and the actual dispatch processing is delayed until a return instruction is issued by the non-task, upon which the actual dispatch processing is performed in batch.

The following shows a processing flow when a service call that involves dispatch processing in a non-task is issued.

Figure 11-5 Delay of Scheduling



## 11.7 Idle Routine

The idle routine is a routine dedicated to idle processing that is extracted as a user-own coding module to utilize the standby function provided by the CPU (to achieve the low-power consumption system), and is called from the scheduler when there no longer remains a task subject to scheduling by the RI78V4 (task in the RUNNING or READY state) in the system.

### 11.7.1 Define idle routine

In the RI78V4, the method of registering an idle routine is limited to "static registration by the Kernel Initialization Module".

Idle routines therefore cannot be created dynamically using a method such as issuing a service call from a processing program.

- Static define
  Static idle routine registration is realized by coding idle routines by using the prescribed function name idle_handler.
  The RI78V4 executes idle routine registration processing based on relevant symbol information, using the Kernel Initialization Module, and handles the registered idle routines as management targets.

### 11.7.2 Undefine idle routine

In the RI78V4, idle routines registered statically by the Kernel Initialization Module cannot be unregistered dynamically using a method such as issuing a service call from a processing program.

### 11.7.3 Basic form of idle routine

Write idle routines using void type functions that do not have arguments (function: idle_handler).
The following shows the basic form of idle routine.

[ C Language ]

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
idle_handler ( void )
{
    /* ............ */          /*Main processing*/

    return;                     /*Terminate idle routine*/
}
```

[ Assembly Language ]

```
$INCLUDE     (kernel.inc)       ;Standard header file definition
$INCLUDE     (kernel_id.inc)    ;System information header file definition

   .PUBLIC  _idle_handler

   .SECTION .textf, TEXTF
_idle_handler:
    ; ............              ;Main processing

    RET                         ;Terminate idle routine
```

### 11.7.4 Internal processing of idle routine

The RI78V4 handles the idle routine as a "non-task (module independent from tasks)".

Moreover, the RI78V4 executes "original pre-processing" when passing control to the idle routine, as well as "original post-processing" when regaining control from the idle routine.

Therefore, note the following points when coding idle routines.

- Coding method
  Code idle routines using C or assembly language in the format shown in "11.7.3 Basic form of idle routine”.

- Stack switching
  The RI78V4 executes processing to switch to the system stack when passing control to the idle routine, and processing to switch to the stack for the switch destination processing program (system stack or task stack) when regaining control from the idle routine.
  The user is therefore not required to code processing related to stack switching in idle routines.

- Interrupt status
  Maskable interrupt acknowledgement is prohibited in the RI78V4 when control is passed to the idle routine.
  The user is therefore not required to write the code related to maskable interrupt acknowledgment in idle routines.

- Service call issuance
  The RI78V4 prohibits issuance of service calls in idle routines.

The following lists processing that should be executed in idle routines.

- Effective use of standby function provided by the CPU

# CHAPTER 12 SERVICE CALLS

This chapter describes the service calls supported by the RI78V4.

## 12.1 Outline

The service calls provided by the RI78V4 are service routines provided for indirectly manipulating the resources (tasks, semaphores, etc.) managed by the RI78V4 from a processing program. The service calls provided by the RI78V4 are listed below by management module.

- Task Management Functions

  act_tsk, iact_tsk, can_act, sta_tsk, ista_tsk, ext_tsk, ter_tsk, chg_pri, ichg_pri, ref_tsk

- Task Dependent Synchronization Functions

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

- Synchronization and Communication Functions (Semaphores)

  sig_sem, isig_sem, wai_sem, pol_sem, twai_sem, ref_sem

- Synchronization and Communication Functions (Eventflags)

  set_flg, iset_flg, clr_flg, wai_flg, pol_flg, twai_flg, ref_flg

- Synchronization and Communication Functions (Data queues)

  snd_dtq, psnd_dtq, ipsnd_dtq, tsnd_dtq, fsnd_dtq, ifsnd_dtq, rcv_dtq, prcv_dtq, trcv_dtq, ref_dtq

- Synchronization and Communication Functions (Mailboxes)

  snd_mbx, rcv_mbx, prcv_mbx, trcv_mbx, ref_mbx

- Memory Pool Management Functions

  get_mpf, pget_mpf, tget_mpf, rel_mpf, ref_mpf

- Time Management Functions

  sta_cyc, stp_cyc, ref_cyc

- System State Management Functions

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

- System Configuration Management Functions

  ref_ver

## 12.2 Call Service Call

The method for calling service calls from processing programs coded either in C or assembly language is described below.

### 12.2.1 C language

By calling using the same method as for normal C functions, service call parameters are handed over to the RI78V4 as arguments and the relevant processing is executed.

[ C Language ]

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void
func_task ( VP_INT exinf )
{
    ER      ercd;               /*Declares variable*/
    ID      tskid = ID_tskA;    /*Declares and initializes variable*/

    ercd = act_tsk ( tskid );   /*Call service call*/

    /* ............ */

    ext_tsk ( );                /*Call service call*/
}
```

Note To call the service calls provided by the RI78V4 from a processing program, the header files listed below must be coded (include processing).

kernel.h: Standard header file (for C language)
kernel_id.h: System information header file (for C language)

### 12.2.2 Assembly language

By calling with the CALL instruction after performing the parameter settings according to the assembler's function calling rules, the service call parameters are handed over to the RI78V4 and the relevant processing is executed.

[ Assembly Language ]

```
$INCLUDE        (kernel.inc)            ;Standard header file definition
$INCLUDE        (kernel_id.inc)         ;System information header file definition

        .SECTION .bss, BSS
_ercd:
        DS      (2)                     ;Secures area for storing return value

       .PUBLIC  _func_task
       .SECTION  .textf, TEXTF
_func_task:
        MOV     A, #ID_tskA             ;Parameter setting
        CALL    !!_act_tsk              ;Call service call
        MOVW    !LOWW(_ercd), AX        ;Return value setting

        ............
        ............

        CALL    !!_ext_tsk              ;Call service call
        BR      !!__kernel_int_exit     ;Jump to end of processing
```

Note To call the service calls provided by the RI78V4 from a processing program, the header files listed below must be coded (include processing).

kernel.inc: Standard header file (for assembly language)
kernel_id.inc: System information header file (for assembly language)

## 12.3 Amount of Stack Used by Service Calls

The RI78V4 saves/restores the values of registers PC, PSW and HL to/from the stack of the processing program that issued the relevant service call (task stack or system stack) during preprocessing/postprocessing of the service call.

The stack of the processing program that issued a service call is used for storing the service call arguments, and the system stack is used as the stack area required for executing internal processing of the service call.

When securing the task stack and system stack areas, the stack amount consumed upon issuance of a service call must therefore be considered.

The following lists the stack sizes required upon issuance of a service call.

- Synchronization and Communication Functions (Data queues)

  snd_dtq, psnd_dtq, ipsnd_dtq, tsnd_dtq, fsnd_dtq, ifsnd_dtq, rcv_dtq, prcv_dtq, prcv_dtq, trcv_dtq, ref_dtq

Table 12-1 Stack Amount Used by Service Call (Unit: Bytes)

| Service Call | For Service Call Arguments | For Internal Processing by Program Issued the Service Call | For System Stack Internal Processing |
|---|---|---|---|
| Task Management Functions | | | |
| act_tsk, iact_tsk | 0 | 10 | 6 |
| can_act | 0 | 10 | 6 |
| sta_tsk, ista_tsk | 0 | 10 | 6 |
| ext_tsk | 0 | 10 | 12 |
| ter_tsk | 0 | 10 | 12 |
| chg_pri, ichg_pri | 0 | 10 | 16 |
| ref_tsk | 0 | 10 | 6 |
| Task Dependent Synchronization Functions | | | |
| slp_tsk | 0 | 10 | 12 |
| tslp_tsk | 0 | 10 | 14 |
| wup_tsk, iwup_tsk | 0 | 10 | 6 |
| can_wup, ican_wup | 0 | 10 | 6 |
| rel_wai, irel_wai | 0 | 10 | 8 |
| sus_tsk, isus_tsk | 0 | 10 | 12 |
| rsm_tsk, irsm_tsk | 0 | 10 | 6 |
| frsm_tsk, ifrsm_tsk | 0 | 10 | 6 |
| dly_tsk | 0 | 10 | 14 |
| Synchronization and Communication Functions (Semaphores) | | | |
| sig_sem, isig_sem | 0 | 10 | 6 |
| wai_sem | 0 | 10 | 12 |
| pol_sem | 0 | 10 | 12 |
| twai_sem | 0 | 10 | 12 |
| ref_sem | 0 | 10 | 6 |
| Synchronization and Communication Functions (Eventflags) | | | |
| set_flg, iset_flg | 0 | 10 | 6 |
| clr_flg | 0 | 10 | 6 |
| wai_flg | 8 | 10 | 16 |
| pol_flg | 0 | 10 | 16 |
| twai_flg | 4 | 10 | 16 |
| ref_flg | 0 | 10 | 6 |
| Synchronization and Communication Functions (Data queues) | | | |
| snd_dtq | 0 | 10 | 16 |
| psnd_dtq, ipsnd_dtq | 0 | 10 | 16 |
| tsnd_dtq | 4 | 10 | 16 |
| snd_mbx | 0 | 10 | 6 |
| fsnd_dtq, ifsnd_dtq | 0 | 10 | 14 |
| rcv_dtq | 0 | 10 | 14 |

| Service Call | For Service Call Arguments | For Internal Processing by Program Issued the Service Call | For System Stack Internal Processing |
|---|---|---|---|
| prcv_dtq, prcv_dtq | 4 | 10 | 14 |
| trcv_dtq | 0 | 10 | 6 |
| ref_dtq | 0 | 10 | 16 |
| Synchronization and Communication Functions (Mailboxes) | | | |
| snd_mbx | 0 | 10 | 8 |
| rcv_mbx | 0 | 10 | 12 |
| prcv_mbx | 0 | 10 | 12 |
| trcv_mbx | 4 | 10 | 12 |
| ref_mbx | 0 | 10 | 6 |
| Memory Pool Management Functions | | | |
| get_mpf | 0 | 10 | 14 |
| pget_mpf | 0 | 10 | 14 |
| tget_mpf | 4 | 10 | 14 |
| rel_mpf | 0 | 10 | 6 |
| ref_mpf | 0 | 10 | 6 |
| Time Management Functions | | | |
| sta_cyc | 0 | 10 | 12 |
| stp_cyc | 0 | 10 | 6 |
| ref_cyc | 0 | 10 | 6 |
| System State Management Functions | | | |
| rot_rdq, irot_rdq | 0 | 10 | 8 |
| get_tid, iget_tid | 0 | 10 | 6 |
| loc_cpu, iloc_cpu | 0 | 10 | 6 |
| unl_cpu, iunl_cpu | 0 | 10 | 8 |
| ena_dsp | 0 | 10 | 6 |
| dis_dsp | 0 | 10 | 6 |
| sns_ctx | 0 | 10 | 6 |
| sns_loc | 0 | 10 | 6 |
| sns_dsp | 0 | 10 | 6 |
| sns_dpn | 0 | 10 | 6 |
| System Configuration Management Functions | | | |
| ref_ver | 0 | 10 | 6 |

## 12.4 Data Macros

This section explains the data macros (for data types, current state, or the like) used when issuing a service call provided by the RI78V4.

### 12.4.1 Data types

The following lists the data types of parameters specified when issuing a service call.

Macro definition of the data type is performed by header file <ri_root>\include\os\types.h, which is called from standard header file <ri_root>\include\kernel.h.

Table 12-2 Data Types

| Macro | Data Type | Description |
|---|---|---|
| UH | unsigned short int | Unsigned 16-bit integer |
| VP | void __near | Pointer to an unknown data type |
| UINT | unsigned int | Unsigned 16-bit integer |
| VP_INT | signed long int | Pointer to an unknown data type, or a signed 32-bit integer |
| ID Note | unsigned char | Object ID number |
| BOOL | signed int | Boolean value |
| STAT | unsigned short int | Object state |
| ER | signed short int | Return value |
| ER_UINT | unsigned short int | Unsigned 16-bit integer |
| PRI | signed char | Priority |
| FLGPTN | unsigned short int | Bit pattern |
| MODE | unsigned char | Service call operational mode |
| TMO | signed long int | Timeout (unit: ticks) |
| RELTIM | unsigned long int | Relative time (unit: ticks) |

Note The ID type definition in the RI78V4 differs from that of the uITRON 4.0 specification.

### 12.4.2 Current state

The following lists the status at the point acquired by issuing a service call (ref_tsk, ref_cyc).
Macro definition of the current status is performed by standard header file <ri_root>\include\kernel.h.

Table 12-3 Current State

| Macro | Value | Description |
|---|---|---|
| TTS_RUN | 0x01 | RUNNING state |
| TTS_RDY | 0x02 | READY state |
| TTS_WAI | 0x04 | WAITING state |
| TTS_SUS | 0x08 | SUSPENDED state |
| TTS_WAS | 0x0c | WAITING-SUSPENDED state |
| TTS_DMT | 0x10 | DORMANT state |
| TCYC_STP | 0x00 | Non-operational state |
| TCYC_STA | 0x01 | Operational state |

### 12.4.3 WAITING types

The following lists WAITING types acquired by issuing a service call (ref_tsk).
Macro definition of the WAITING type is performed by standard header file <ri_root>\include\kernel.h.

Table 12-4 WAITING Types

| Macro | Value | Description |
|---|---|---|
| TTW_SLP | 0x0001 | A task enters this state if the counter for the task (registering the number of times the wakeup request has been issued) indicates 0x0 upon the issuance of a slp_tsk or tslp_tsk. |
| TTW_DLY | 0x0002 | A task enters this state upon the issuance of a dly_tsk. |
| TTW_SEM | 0x0004 | A task enters this state if it cannot acquire a resource from the relevant semaphore upon the issuance of a wai_sem or twai_sem. |
| TTW_FLG | 0x0008 | A task enters this state if a relevant eventflag does not satisfy a predetermined condition upon the issuance of a wai_flg or twai_flg. |
| TTW_SDTQ | 0x0010 | A task enters this state if cannot send a data to the relevant data queue upon the issuance of a snd_dtq or tsnd_dtq. |
| TTW_RDTQ | 0x0020 | A task enters this state if cannot receive a data from the relevant data queue upon the issuance of a rcv_dtq or trcv_dtq. |
| TTW_MBX | 0x0040 | A task enters this state if cannot receive a message from the relevant mailbox upon the issuance of a rcv_mbx or trcv_mbx. |

| Macro | Value | Description |
|---|---|---|
| TTW_MPF | 0x2000 | A task enters this state if it cannot acquire a fixed-sized memory block from the relevant fixed-sized memory pool upon the issuance of a get_mpf or tget_mpf. |

### 12.4.4 Return value

The following lists the values returned from service calls.
Macro definition of the return value is performed by standard header file <ri_root>\include\kernel.h.

Table 12-5 Return Value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_ILUSE | -28 | Illegal service call use. |
| E_OBJ | -41 | Object state error. |
| E_QOVR | -43 | Queue overflow. |
| E_RLWAI | -49 | Forced release from waiting (accept rel_wai/irel_wai while waiting). |
| E_TMOUT | -50 | Polling failure or timeout. |
| FALSE | 0 | False |
| TRUE | 1 | True |

### 12.4.5 Conditional compile macro

The RI78V4 header files are conditionally compiled by the following macro.

Table 12-6 Conditional Compile Macro

| Classification | Macro | Description |
|---|---|---|
| C compiler package | __REL__ | The CC-RL is used. |

### 12.4.6 Others

The following lists other macros used when issuing a service call.
Macro definition of other macros is performed by standard header file <ri_root>\include\kernel.h.

Table 12-7 Others

| Macro | Value | Description |
|---|---|---|
| TSK_SELF | 0 | Invoking task |
| TPRI_INI | 0 | Initial priority of the task |
| TMO_FEVR | -1 | Waiting forever |
| TMO_POL | 0 | Polling |
| TWF_ANDW | 0x00 | AND waiting condition |
| TWF_ORW | 0x01 | OR waiting condition |
| TPRI_SELF | 0 | Current priority of the invoking task |
| TSK_NONE | 0 | No applicable task |
| NULL | 0 | No applicable message |

## 12.5 Packet Formats

This section explains the data structures (task state packet, semaphore state packet, or the like) used when issuing a service call provided by the RI78V4.

### 12.5.1 Task state packet

The following shows task state packet T_RTSK used when issuing ref_tsk.
Definition of task state packet T_RTSK is performed by header file <ri_root>\include\os\{packet.h, packet.inc}, which is called from standard header file <ri_root>\include\{kernel.h, kernel.inc}.

[ packet.h ]

```
typedef struct  t_rtsk {
    STAT    tskstat;        /*Task current state*/
    PRI     tskpri;         /*Task current priority*/
    PRI     tskbpri;        /*Reserved for future use*/
    STAT    tskwait;        /*Reason for waiting*/
    ID      wobjid;         /*Object ID number for which the task is waiting*/
    TMO     lefttmo;        /*Reserved for future use*/
    UINT    actcnt;         /*Activation request count*/
    UINT    wupcnt;         /*Wakeup request count*/
    UINT    suscnt;         /*Suspension count*/
} T_RTSK;
```

[ packet.inc ]

```
rtsk_tskstat    .EQU   0x00   ;Task current state
rtsk_tskpri     .EQU   0x02   ;Task current priority
rtsk_tskbpri    .EQU   0x03   ;Reserved for future use
rtsk_tskwait    .EQU   0x04   ;Reason for waiting
rtsk_wobjid     .EQU   0x06   ;Object ID number for which the task is waiting
rtsk_lefttmo    .EQU   0x08   ;Reserved for future use
rtsk_actcnt     .EQU   0x0c   ;Activation request count
rtsk_wupcnt     .EQU   0x0e   ;Wakeup request count
rtsk_suscnt     .EQU   0x10   ;Suspension count
```

The following shows details on task state packet T_RTSK.

- tskstat, rtsk_tskstat
  Stores the current state of the task.

  TTS_RUN: RUNNING state
  TTS_RDY: READY state
  TTS_WAI: WAITING state
  TTS_SUS: SUSPENDED state
  TTS_WAS: WAITING-SUSPENDED state
  TTS_DMT: DORMANT state

- tskpri, rtsk_tskpri
  Stores the current priority of the task.

- tskbpri, rtsk_tskbpri
  System-reserved area.

- tskwait, rtsk_tskwait
  Stores the reason for waiting.

  TTW_NONE: Has not moved to the WAITING state.

TTW_SLP: A task enters this state if the counter for the task (registering the number of times the wakeup request has been issued) indicates 0x0 upon the issuance of a slp_tsk or tslp_tsk.

TTW_DLY: A task enters this state upon the issuance of a dly_tsk.

TTW_SEM: A task enters this state if it cannot acquire a resource from the relevant semaphore upon the issuance of a wai_sem or twai_sem.

TTW_FLG: A task enters this state if a relevant eventflag does not satisfy a predetermined condition upon the issuance of a wai_flg or twai_flg.

TTW_SDTQ: A task enters this state if cannot send a data to the relevant data queue upon the issuance of a snd_dtq or tsnd_dtq.

TTW_RDTQ: A task enters this state if cannot receive a data from the relevant data queue upon the issuance of a rcv_dtq or trcv_dtq.

TTW_MBX: A task enters this state if cannot receive a message from the relevant mailbox upon the issuance of a rcv_mbx or trcv_mbx.

TTW_MPF: A task enters this state if it cannot acquire a fixed-sized memory block from the relevant fixed-sized memory pool upon the issuance of a get_mpf or tget_mpf.

- wobjid, rtsk_wobjid
  Stores the object ID number for which the task is waiting.

- lefttmo, rtsk_lefttmo
  System-reserved area.

- actcnt, rtsk_actcnt
  Stores the activation request count of the task.

- wupcnt, rtsk_wupcnt
  Stores the wakeup request count of the task.

- suscnt, rtsk_suscnt
  Stores the suspention count of the task.

### 12.5.2 Semaphore state packet

The following shows semaphore state packet T_RSEM used when issuing ref_sem.
Definition of semaphore state packet T_RSEM is performed by header file <ri_root>\include\os\{packet.h, packet.inc}, which is called from standard header file <ri_root>\include\{kernel.h, kernel.inc}.

[ packet.h ]

```
typedef struct  t_rsem {
    ID      wtskid;         /*ID number of the task at the head of the wait queue*/
    UINT    semcnt;         /*Current resource count*/
} T_RSEM;
```

[ packet.inc ]

```
rsem_wtskid     .EQU   0x00   ;ID number of the task at the head of the wait queue
rsem_semcnt     .EQU   0x02   ;Current resource count
```

The following shows details on semaphore state packet T_RSEM.

- wtskid, rsem_wtskid
  Stores information whether a task is queued to the wait queue.

  TSK_NONE: No applicable task.
  Value: ID number of the task at the head of the wait queue

- semcnt, rsem_semcnt
  Stores the current resource count of the semaphore.

### 12.5.3 Eventflag state packet

The following shows eventflag state packet T_RFLG used when issuing ref_flg.

Definition of eventflag state packet T_RFLG is performed by header file <ri_root>\include\os\{packet.h, packet.inc}, which is called from standard header file <ri_root>\include\{kernel.h, kernel.inc}.

[ packet.h ]

```
typedef struct  t_rflg {
    ID      wtskid;         /*ID number of the task at the head of the wait queue*/
    FLGPTN  flgptn;         /*Current bit pattern*/
} T_RFLG;
```

[ packet.inc ]

```
rflg_wtskid     .EQU   0x00   ;ID number of the task at the head of the wait queue
rflg_flgptn     .EQU   0x02   ;Current bit pattern
```

The following shows details on eventflag state packet T_RFLG.

- wtskid, rflg_wtskid
  Stores information whether a task is queued to the wait queue.

  TSK_NONE: No applicable task.
  Value: ID number of the task at the head of the wait queue

- flgptn, rflg_flgptn
  Stores the current bit pattern of the eventflag.

### 12.5.4 Data queue state packet

The following shows data queue state packet T_RDTQ used when issuing ref_dtq or iref_dtq.
Definition of data queue state packet T_RDTQ is performed by header file <ri_root>\include\os\{packet.h, packet.inc}, which is called from standard header file <ri_root>\include\{kernel.h, kernel.inc}.

【 packet.h 】

```
typedef struct  t_rdtq {
    ID      stskid;         /*Existence of tasks waiting for data transmission*/
    ID      rtskid;         /*Existence of tasks waiting for data reception*/
    UINT    sdtqcnt;        /*number of data elements in the data queue*/
} T_RDTQ;
```

【 packet.inc 】

```
rdtq_stskid     .EQU     0x00   ; Existence of tasks waiting for data transmission
rdtq_rtskid     .EQU     0x01   ; Existence of tasks waiting for data reception
rdtq_sdtqcnt    .EQU     0x02   ; number of data elements in the data queue
```

The following shows details on data queue state packet T_RDTQ.

- stskid
  Stores whether a task is queued to the transmission wait queue of the data queue.

  TSK_NONE: No applicable task
  Value: ID number of the task at the head of the wait queue

- rtskid
  Stores whether a task is queued to the reception wait queue of the data queue.

  TSK_NONE: No applicable task
  Value: ID number of the task at the head of the wait queue

- sdtqcnt
  Stores the number of data elements in data queue.

### 12.5.5 Message packet

The following shows message packet T_MSG and T_MSG_PRI used when issuing snd_mbx, rcv_mbx, prcv_mbx, or trcv_mbx.

Definition of message packet T_MSG and T_MSG_PRI is performed by header file <ri_root>\include\types.h, which is called from standard header file <ri_root>\include\kernel.h.

[ Message packet for TA_MFIFO attribute ]

```
typedef struct  t_msg {
    struct  t_msg   __near   *msgque;  /*Reserved for future use*/
} T_MSG;
```

[ Message packet for TA_MPRI attribute ]

```
typedef struct  t_msg_pri {
    struct  t_msg   __near   *msgque;  /*Reserved for future use*/
    PRI     msgpri;                    /*Message priority*/
} T_MSG_PRI;
```

The following shows details on message packet T_MSG and T_MSG_PRI.

- msgque
  System-reserved area.

- msgpri
  Stores the priority of the message.

  Note 1 In the RI78V4, a message having a smaller priority number is given a higher priority.

  Note 2 Values that can be specified for the priority of a message are limited from 1 to 31.

### 12.5.6 Mailbox state packet

The following shows mailbox state packet T_RMBX used when issuing ref_mbx.

Definition of mailbox state packet T_RMBX is performed by header file <ri_root>\include\os\{packet.h, packet.inc}, which is called from standard header file <ri_root>\include\{kernel.h, kernel.inc}.

[ packet.h ]

```
typedef struct  t_rmbx {
    ID      wtskid;             /*ID number of the task at the head of the wait
                                  queue*/
    T_MSG   __near   *pk_msg;   /*Start address of the message packet at the head
                                  of the message queue*/
} T_RMBX;
```

[ packet.inc ]

```
rmbx_wtskid     .EQU   0x00     ;ID number of the task at the head of the wait
                                ;queue
rmbx_pk_msg     .EQU   0x02     ;Start address of the message packet at the head
                                ;of the message queue
```

The following shows details on mailbox state packet T_RMBX.

- wtskid, rmbx_wtskid
  Stores information whether a task is queued to the wait queue.

  TSK_NONE: No applicable task.
  Value: ID number of the task at the head of the wait queue

- pk_msg, rmbx_pk_msg
  Stores information whether a message is queued to the message queue.

  NULL: No applicable message.
  Value: Start address of the message packet at the head of the message queue

### 12.5.7 Fixed-sized memory pool state packet

The following shows fixed-sized memory pool state packet T_RMPF used when issuing ref_mpf.

Definition of fixed-sized memory pool state packet T_RMPF is performed by header file <ri_root>\include\os\{packet.h, packet.inc}, which is called from standard header file <ri_root>\include\{kernel.h, kernel.inc}.

[ packet.h ]

```
typedef struct  t_rmpf {
    ID      wtskid;         /*ID number of the task at the head of the wait queue*/
    UINT    fblkcnt;        /*Number of free memory blocks*/
} T_RMPF;
```

[ packet.inc ]

```
rmpf_wtskid     .EQU   0x00   ;ID number of the task at the head of the wait queue
rmpf_fblkcnt    .EQU   0x02   ;Number of free memory blocks
```

The following shows details on fixed-sized memory pool state packet T_RMPF.

- wtskid, rmpf_wtskid
  Stores information whether a task is queued to the wait queue.

  TSK_NONE: No applicable task.
  Value: ID number of the task at the head of the wait queue

- fblkcnt, rmpf_fblkcnt
  Stores the number of free memory blocks.

### 12.5.8 Cyclic handler state packet

The following shows cyclic handler state packet T_RCYC used when issuing ref_cyc.

Definition of cyclic handler state packet T_RCYC is performed by header file <ri_root>\include\os\{packet.h, packet.inc}, which is called from standard header file <ri_root>\include\{kernel.h, kernel.inc}.

[ packet.h ]

```
typedef struct  t_rcyc {
    STAT    cycstat;        /*Cyclic handler operational state*/
    RELTIM  lefttim;        /*Time left before the next activation*/
} T_RCYC;
```

[ packet.inc ]

```
rcyc_cycstat    .EQU   0x00   ;Cyclic handler operational state
rcyc_lefttim    .EQU   0x02   ;Time left before the next activation
```

The following shows details on cyclic handler state packet T_RCYC.

- cycstat, rcyc_cycstat
  Stores the operational state of the cyclic handler.

  TCYC_STP: Operational state
  TCYC_STA: Non-operational state

- lefttim, rcyc_lefttim
  Stores the time (unit: tick) left before the next activation.
  The contents of this member become an undefined value if the target cyclic handler is in the non-operational state (STP state).

### 12.5.9 Version information packet

The following shows version information packet T_RVER used when issuing ref_ver.

Definition of version information packet T_RVER is performed by header file <ri_root>\include\os\{packet.h, packet.inc}, which is called from standard header file <ri_root>\include\{kernel.h, kernel.inc}.

[ packet.h ]

```
typedef struct  t_rver {
    UH      maker;          /*Kernel maker's code*/
    UH      prid;           /*Identification number of the kernel*/
    UH      spver;          /*Version number of the ITRON Specification*/
    UH      prver;          /*Version number of the kernel*/
    UH      prno[4];        /*Management information of the kernel product*/
} T_RVER;
```

[ packet.inc ]

```
verinf_maker    .EQU   0x00   ;Kernel maker's code
verinf_prid     .EQU   0x02   ;Identification number of the kernel
verinf_spver    .EQU   0x04   ;Version number of the ITRON Specification
verinf_prver    .EQU   0x06   ;Version number of the kernel
verinf_prno     .EQU   0x08   ;Management information of the kernel product
```

The following shows details on version information packet T_RVER.

- maker, verinf_maker
  Stores the kernel maker's code.
  0x011b: Renesas Electronics Co., Ltd.

- prid, verinf_prid
  Stores the identification number of the kernel.
  0x0006: Identification number

- spver, verinf_spver
  Stores the version number of the ITRON Specification.
  0x5403: μITRON4.0 Specification Ver.4.03.00

- prver, verinf_prver
  Stores the version number of the kernel.
  0x01*xx*: Ver.2.*xx*

- prno[0], verinf_prno
  Stores the kernel version type.
  0x0: V-version

- prno[1], verinf_prno + 0x2
  Stores the memory model of the kernel.
  0x2 Medium model

- prno[2], verinf_prno + 0x4
  System-reserved area.

- prno[3], verinf_prno + 0x6
  System-reserved area.

## 12.6 Task Management Functions

The following lists the service calls provided by the RI78V4 as the task management functions.

Table 12-8 Task Management Functios

| Service Call | Function | Origin of Service Call |
|---|---|---|
| act_tsk | Activate task (queues an activation request). | Task, Non-task |
| iact_tsk | Activate task (queues an activation request). | Task, Non-task |
| can_act | Cancel task activation requests. | Task, Non-task |
| sta_tsk | Activate task (does not queue an activation request). | Task, Non-task |
| ista_tsk | Activate task (does not queue an activation request). | Task, Non-task |
| ext_tsk | Terminate invoking task. | Task |
| ter_tsk | Terminate task. | Task |
| chg_pri | Change task priority. | Task, Non-task |
| ichg_pri | Change task priority. | Task, Non-task |
| ref_tsk | Reference task state. | Task, Non-task |

# act_tsk
# iact_tsk

## Outline

Activate task (queues an activation request).

## C format

```
ER      act_tsk ( ID tskid );

ER      iact_tsk ( ID tskid );
```

## Assembly format

```
MOV     A, #tskid
CALL    !!_act_tsk

MOV     A, #tskid
CALL    !!_iact_tsk
```

## Parameter(s)

| I/O | Parameter | Description |
| --- | --- | --- |
| I | ID *tskid*; | ID number of the task to be activated.<br>TSK_SELF: Invoking task.<br>Value: ID number of the task to be activated. |

## Explanation

These service calls move a task specified by parameter *tskid* from the DORMANT state to the READY state.

As a result, the target task is queued at the end on the ready queue corresponding to the initial priority and becomes subject to scheduling by the RI78V4.

If the target task has been moved to a state other than the DORMANT state when this service call is issued, this service call does not move the state but increments the activation request counter (by added 0x1 to the wakeup request counter).

Note 1 The activation request counter managed by the RI78V4 is configured in 7-bit widths. If the number of activation requests exceeds the maximum count value 127 as a result of issuing this service call, the counter manipulation processing is therefore not performed but "E_QOVR" is returned.

Note 2 An extended infomration "Extended information: exinf" is passed to the task activated by issuing this service call.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_QOVR | -43 | Queue overflow (overflow of activation request count "127"). |

# can_act

## Outline

Cancel task activation requests.

## C format

```
ER_UINT can_act ( ID tskid );
```

## Assembly format

```
MOV     A, #tskid
CALL    !!_can_act
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID tskid;` | ID number of the task for cancelling activation requests.<br>TSK_SELF: Invoking task.<br>Value: ID number of the task for cancelling activation requests. |

## Explanation

This service call cancels all of the activation requests queued to the task specified by parameter *tskid* (sets the activation request counter to 0x0).

When this service call is terminated normally, the number of cancelled activation requests is returned.

## Return value

| Macro | Value | Description |
|---|---|---|
| - | - | Normal completion (activation request count: positive value or 0). |

# sta_tsk
# ista_tsk

## Outline

Activate task (does not queue an activation request).

## C format

```
ER      sta_tsk ( ID tskid, VP_INT stacd );

ER      ista_tsk ( ID tskid, VP_INT stacd );
```

## Assembly format

```
MOVW    BC, #stacd_lo
MOVW    DE, #stacd_hi
MOV     A, #tskid
CALL    !!_sta_tsk

MOVW    BC, #stacd_lo
MOVW    DE, #stacd_hi
MOV     A, #tskid
CALL    !!_ista_tsk
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID tskid;` | ID number of the task to be activated. |
| I | `VP_INT stacd;` | Start code of the task. |

## Explanation

These service calls move a task specified by parameter *tskid* from the DORMANT state to the READY state.

As a result, the target task is queued at the end on the ready queue corresponding to the initial priority and becomes subject to scheduling by the RI78V4.

Note 1 This service call does not perform queuing of activation requests. If the target task is in a state other than the DORMANT state, the counter manipulation processing is therefore not performed but "E_OBJ" is returned.

Note 2 A start code "*stacd*" is passed to the task activated by issuing this service call.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

| Macro | Value | Description |
|---|---|---|
| E_OBJ | -41 | Object state error (specified task is not in the DORMANT state). |

# ext_tsk

## Outline

Terminate invoking task.

## C format

```
void    ext_tsk ( void );
```

## Assembly format

```
BR      !!_ext_tsk
```

## Parameter(s)

None.

## Explanation

This service call moves an invoking task from the RUNNING state to the DORMANT state.

As a result, the invoking task is unlinked from the ready queue and excluded from the RI78V4 scheduling subject.

If an activation request has been queued to the invoking task (the activation request counter is not set to 0x0) when this service call is issued, this service call moves the task from the RUNNING state to the DORMANT state, decrements the wakeup request counter (by subtracting 0x1 from the wakeup request counter), and then moves the task from the DORMANT state to the READY state.

Note 1 This service call does not return the OS resource that the invoking task acquired by issuing a service call such as sig_sem or get_mpf. The OS resource have been acquired must therefore be returned before issuing this service call.

Note 2 When moving a task from the RUNNING state to the DORMANT state, this service call initializes the following information to values that are set during task creation.

- Priority (current priority)
- Wakeup request count
- Suspension count
- Interrupt status

Note 3 If the return instruction is written in a task, it executes the same operation as this service call.

Note 4 In the RI78V4, code efficiency is enhanced by coding the return instruction as a "Terminate invoking task".

## Return value

None.

# ter_tsk

## Outline

Terminate task.

## C format

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

## Assembly format

```
MOV     A, #tskid
CALL    !!_ter_tsk
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID tskid;` | ID number of the task to be terminated. |

## Explanation

This service call forcibly moves a task specified by parameter *tskid* to the DORMANT state.

As a result, the target task is excluded from the RI78V4 scheduling subject.

If an activation request has been queued to the target task (the activation request counter is not set to 0x0) when this service call is issued, this service call moves the task to the DORMANT state, decrements the wakeup request counter (by subtracting 0x1 from the wakeup request counter), and then moves the task from the DORMANT state to the READY state.

Note 1 This service call does not return the OS resource that the target task acquired by issuing a service call such as sig_sem or get_mpf. The OS resource have been acquired must therefore be returned before issuing this service call.

Note 2 When moving a task to the DORMANT state, this service call initializes the following information to values that are set during task creation.

- Priority (current priority)
- Wakeup request count
- Suspension count
- Interrupt status

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_OBJ | -41 | Object state error (specified task is in the DORMANT state). |

# chg_pri
# ichg_pri

## Outline

Change task priority.

## C format

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

ER      ichg_pri ( ID tskid, PRI tskpri );
```

## Assembly format

```
MOVW    AX, #(tskid | tskpri)
CALL    !!_chg_pri

MOVW    AX, #(tskid | tskpri)
CALL    !!_ichg_pri
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | ID *tskid*; | ID number of the task whose priority is to be changed.<br>TSK_SELF: Invoking task.<br>Value: ID number of the task whose priority is to be changed. |
| I | PRI *tskpri*; | New current priority of the task.<br>TPRI_INI: Initial priority of the task.<br>Value: New current priority of the task. |

## Explanation

These service calls change the priority of the task specified by parameter *tskid* (current priority) to a value specified by parameter *tskpri*.

Note If the target task is in the RUNNING or READY state after this service call is issued, this service call re-queues the task at the end of the ready queue corresponding to the priority specified by parameter *tskpri*, following priority change processing.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

| Macro | Value | Description |
| --- | --- | --- |
| E_OBJ | -41 | Object state error (specified task is in the DORMANT state). |

# ref_tsk

## Outline

Reference task state.

## C format

```
ER      ref_tsk ( ID tskid, T_RTSK *pk_rtsk );
```

## Assembly format

```
MOVW    BC, #LOWW(_pk_rtsk)
MOV     A,  #tskid
CALL    !!_ref_tsk
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID tskid;` | ID number of the task to be referenced.<br>TSK_SELF: Invoking task.<br>Value: ID number of the task to be referenced. |
| O | `T_RTSK *pk_rtsk;` | Pointer to the packet returning the task state. |

## Explanation

Stores task state packet (such as current status) of the task specified by parameter *tskid* in the area specified by parameter *pk_rtsk*.

Note For details about the task state packet, refer to "12.5.1 Task state packet".

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

## 12.7 Task Dependent Synchronization Functions

The following lists the service calls provided by the RI78V4 as the task dependent synchronization functions.

Table 12-9 Task Dependent Synchronization Functions

| Service Call | Function | Origin of Service Call |
|---|---|---|
| slp_tsk | Put task to sleep (waiting forever). | Task |
| tslp_tsk | Put task to sleep (with timeout). | Task |
| wup_tsk | Wakeup task. | Task, Non-task |
| iwup_tsk | Wakeup task. | Task, Non-task |
| can_wup | Cancel task wakeup requests. | Task, Non-task |
| ican_wup | Cancel task wakeup requests. | Task, Non-task |
| rel_wai | Release task from waiting. | Task, Non-task |
| irel_wai | Release task from waiting. | Task, Non-task |
| sus_tsk | Suspend task. | Task, Non-task |
| isus_tsk | Suspend task. | Task, Non-task |
| rsm_tsk | Resume suspended task. | Task, Non-task |
| irsm_tsk | Resume suspended task. | Task, Non-task |
| frsm_tsk | Forcibly resume suspended task. | Task, Non-task |
| ifrsm_tsk | Forcibly resume suspended task. | Task, Non-task |
| dly_tsk | Delay task. | Task |

# slp_tsk

## Outline

Put task to sleep (waiting forever).

## C format

```
ER      slp_tsk ( void );
```

## Assembly format

```
CALL    !!_slp_tsk
```

## Parameter(s)

None.

## Explanation

As a result, the invoking task is unlinked from the ready queue and excluded from the RI78V4 scheduling subject.

If a wakeup request has been queued to the target task (the wakeup request counter is not set to 0x0) when this service call is issued, this service call does not move the state but decrements the wakeup request counter (by subtracting 0x1 from the wakeup request counter).

| Sleeping State Cancel Operation | Return Value |
|---|---|
| A wakeup request was issued as a result of issuing wup_tsk. | E_OK |
| A wakeup request was issued as a result of issuing iwup_tsk. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_RLWAI | -49 | Forced release from waiting (accept rel_wai/irel_wai while waiting). |

# tslp_tsk

## Outline

Put task to sleep (with timeout).

## C format

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

## Assembly format

```
MOVW    BC, #tmout_hi
MOVW    AX, #tmout_lo
CALL    !!_tslp_tsk
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | TMO *tmout;* | Specified timeout (unit: ticks).<br>TMO_FEVR: Waiting forever.<br>TMO_POL: Polling.<br>Value: Specified timeout. |

## Explanation

This service call moves an invoking task from the RUNNING state to the WAITING state (sleeping state).

As a result, the invoking task is unlinked from the ready queue and excluded from the RI78V4 scheduling subject.

If a wakeup request has been queued to the target task (the wakeup request counter is not set to 0x0) when this service call is issued, this service call does not move the state but decrements the wakeup request counter (by subtracting 0x1 from the wakeup request counter).

The sleeping state is cancelled in the following cases, and then moved to the READY state.

| Sleeping State Cancel Operation | Return Value |
|---|---|
| A wakeup request was issued as a result of issuing wup_tsk. | E_OK |
| A wakeup request was issued as a result of issuing iwup_tsk. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |
| Polling failure or timeout. | E_TMOUT |

Note    When TMO_FEVR is specified for wait time *tmout*, processing equivalent to slp_tsk will be executed.

### Return value

| Macro | Value | Description |
| --- | --- | --- |
| E_OK | 0 | Normal completion. |
| E_RLWAI | -49 | Forced release from waiting (accept rel_wai/irel_wai while waiting). |
| E_TMOUT | -50 | Polling failure or timeout. |

# wup_tsk
# iwup_tsk

## Outline

Wakeup task.

## C format

```
ER      wup_tsk ( ID tskid );

ER      iwup_tsk ( ID tskid );
```

## Assembly format

```
MOV     A, #tskid
CALL    !!_wup_tsk

MOV     A, #tskid
CALL    !!_iwup_tsk
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | ID *tskid*; | ID number of the task to be woken up.<br>TSK_SELF: Invoking task.<br>Value: ID number of the task to be woken up. |

## Explanation

These service calls cancel the WAITING state (sleeping state) of the task specified by parameter *tskid*.

As a result, the target task is moved from the sleeping state to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

If the target task is in a state other than the sleeping state when this service call is issued, this service call does not move the state but increments the wakeup request counter (by added 0x1 to the wakeup request counter).

Note 1 If the target task is moved to the READY state after this service call is issued, this service call also re-queues the task at the end of the ready queue corresponding to the priority of the task.

Note 2 The wakeup request counter managed by the RI78V4 is configured in 7-bit widths. If the number of wakeup requests exceeds the maximum count value 127 as a result of issuing this service call, the counter manipulation processing is therefore not performed but "E_QOVR" is returned.

### Return value

| Macro | Value | Description |
| --- | --- | --- |
| E_OK | 0 | Normal completion. |
| E_OBJ | -41 | Object state error (specified task is in the DORMANT state). |
| E_QOVR | -43 | Queue overflow (overflow of wakeup request count "127"). |

# can_wup
# ican_wup

## Outline

Cancel task wakeup requests.

## C format

```
ER_UINT can_wup ( ID tskid );

ER_UINT ican_wup ( ID tskid );
```

## Assembly format

```
MOV     A, #tskid
CALL    !!_can_wup

MOV     A, #tskid
CALL    !!_ican_wup
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | ID *tskid*; | ID number of the task for cancelling wakeup requests.<br>TSK_SELF: Invoking task.<br>Value: ID number of the task for cancelling wakeup requests. |

## Explanation

These service calls cancel all of the wakeup requests queued to the task specified by parameter *tskid* (the wakeup request counter is set to 0x0).

When this service call is terminated normally, the number of cancelled wakeup requests is returned.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OBJ | -41 | Object state error (specified task is in the DORMANT state). |
| - | - | Normal completion (wakeup request count: positive value or 0). |

# rel_wai
# irel_wai

## Outline

Release task from waiting.

## C format

```
ER      rel_wai ( ID tskid );

ER      irel_wai ( ID tskid );
```

## Assembly format

```
MOV     A, #tskid
CALL    !!_rel_wai

MOV     A, #tskid
CALL    !!_irel_wai
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | ID *tskid*; | ID number of the task to be released from waiting. |

## Explanation

These service calls forcibly cancel the WAITING state of the task specified by parameter *tskid*.

As a result, the target task unlinked from the wait queue and is moved from the WAITING state to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

"E_RLWAI" is returned from the service call that triggered the move to the WAITING state (slp_tsk, wai_sem, or the like) to the task whose WAITING state is cancelled by this service call.

Note 1 If the target task is moved to the READY state after this service call is issued, this service call also re-queues the task at the end of the ready queue corresponding to the priority of the task.

Note 2 This service call does not perform queuing of forced cancellation requests. If the target task is in a state other than the WAITING or WAITING-SUSPENDED state, "E_OBJ" is returned.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_OBJ | -41 | Object state error (specified task is neither in the WAITING state nor WAITING-SUSPENDED state). |

# sus_tsk
# isus_tsk

## Outline

Suspend task.

## C format

```
ER      sus_tsk ( ID tskid );

ER      isus_tsk ( ID tskid );
```

## Assembly format

```
MOV     A, #tskid
CALL    !!_sus_tsk

MOV     A, #tskid
CALL    !!_isus_tsk
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | ID *tskid*; | ID number of the task to be suspended.<br>TSK_SELF: Invoking task.<br>Value: ID number of the task to be suspended. |

## Explanation

These service calls add 0x1 to the suspend request counter for the task specified by parameter *tskid*, and then move the target task from the RUNNING state to the SUSPENDED state, from the READY state to the SUSPENDED state, or from the WAITING state to the WAITING-SUSPENDED state.

If the target task has moved to the SUSPENDED or WAITING-SUSPENDED state when this service call is issued, the counter manipulation processing is not performed but only the suspend request counter increment processing is executed.

| SUSPENDED State Cancel Operation | Return Value |
|---|---|
| A cancel request was issued as a result of issuing rsm_tsk. | E_OK |
| A cancel request was issued as a result of issuing irsm_tsk. | E_OK |
| Forced release from suspended (accept frsm_tsk while suspended). | E_OK |
| Forced release from suspended (accept ifrsm_tsk while suspended). | E_OK |

Note 1 If the target task is the invoking task when this service call is issued, it is unlinked from the ready queue and excluded from the RI78V4 scheduling subject.

Note 2 The suspend request counter managed by the RI78V4 is configured in 7-bit widths. If the number of suspend requests exceeds the maximum count value 127 as a result of issuing this service call, the counter manipulation processing is therefore not performed but "E_QOVR" is returned.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_OBJ | -41 | Object state error (specified task is in the DORMANT state). |
| E_QOVR | -43 | Queue overflow (overflow of suspension count "127"). |

# rsm_tsk
# irsm_tsk

## Outline

Resume suspended task.

## C format

```
ER      rsm_tsk ( ID tskid );

ER      irsm_tsk ( ID tskid );
```

## Assembly format

```
MOV     A, #tskid
CALL    !!_rsm_tsk

MOV     A, #tskid
CALL    !!_irsm_tsk
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | ID *tskid*; | ID number of the task to be resumed. |

## Explanation

This service call subtracts 0x1 from the suspend request counter for the task specified by parameter *tskid*, and then cancels the SUSPENDED state of the target task.

As a result, the target task is moved from the SUSPENDED state to the READY state, or from the WAITING-SUSPENDED state to the WAITING state.

If a suspend request is queued (subtraction result is other than 0x0) when this service call is issued, the counter manipulation processing is not performed but only the suspend request counter decrement processing is executed.

Note 1 If the target task is moved to the READY state after this service call is issued, this service call also re-queues the task at the end of the ready queue corresponding to the priority of the task.

Note 2 This service call does not perform queuing of cancellation requests. If the target task is in a state other than the SUSPENDED or WAITING-SUSPENDED state, "E_OBJ" is therefore returned.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

| Macro | Value | Description |
|---|---|---|
| E_OBJ | -41 | Object state error (specified task is neither in the SUSPENDED state nor WAITING-SUSPENDED state). |

# frsm_tsk
# ifrsm_tsk

## Outline

Forcibly resume suspended task.

## C format

```
ER      frsm_tsk ( ID tskid );

ER      ifrsm_tsk ( ID tskid );
```

## Assembly format

```
MOV     A, #tskid
CALL    !!_frsm_tsk

MOV     A, #tskid
CALL    !!_ifrsm_tsk
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID` *tskid*; | ID number of the task to be resumed. |

## Explanation

These service calls set the suspend request counter for the task specified by parameter *tskid* to 0x1 f, and then forcibly cancel the SUSPENDED state of the target task.

As a result, the target task is moved from the SUSPENDED state to the READY state, or from the WAITING-SUSPENDED state to the WAITING state.

Note 1 If the target task is moved to the READY state after this service call is issued, this service call also re-queues the task at the end of the ready queue corresponding to the priority of the task.

Note 2 This service call does not perform queuing of forced cancellation requests. If the target task is in a state other than the SUSPENDED or WAITING-SUSPENDED state, "E_OBJ" is therefore returned.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_OBJ | -41 | Object state error (specified task is neither in the SUSPENDED state nor WAITING-SUSPENDED state). |

# dly_tsk

## Outline

Delay task.

## C format

```
ER      dly_tsk ( RELTIM dlytim );
```

## Assembly format

```
MOVW    BC, #dlytim_hi
MOVW    AX, #dlytim_lo
CALL    !!_dly_tsk
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | RELTIM *dlytim*; | Amount of relative time to delay the invoking task (unit: ticks). |

## Explanation

This service call moves the invoking task from the RUNNING state to the WAITING state (delayed state).
As a result, the invoking task is unlinked from the ready queue and excluded from the RI78V4 scheduling subject.
The delayed state is cancelled in the following cases, and then moved to the READY state.

| Delayed State Cancel Operation | Return Value |
|---|---|
| Delay time specified by parameter *dlytim* has elapsed. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_RLWAI | -49 | Forced release from waiting (accept rel_wai/irel_wai while waiting). |

## 12.8 Synchronization and Communication Functions (Semaphores)

The following lists the service calls provided by the RI78V4 as the synchronization and communication functions (semaphores).

Table 12-10 Synchronization and Communication Functions (Semaphores)

| Service Call | Function | Origin of Service Call |
|---|---|---|
| sig_sem | Release semaphore resource. | Task, Non-task |
| isig_sem | Release semaphore resource. | Task, Non-task |
| wai_sem | Acquire semaphore resource (waiting forever). | Task |
| pol_sem | Acquire semaphore resource (polling). | Task, Non-task |
| twai_sem | Acquire semaphore resource (with timeout). | Task |
| ref_sem | Reference semaphore state. | Task, Non-task |

# sig_sem
# isig_sem

## Outline

Release semaphore resource.

## C format

```
ER      sig_sem ( ID semid );

ER      isig_sem ( ID semid );
```

## Assembly format

```
MOV     A, #semid
CALL    !!_sig_sem

MOV     A, #semid
CALL    !!_isig_sem
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | ID *semid*; | ID number of the semaphore to which resource is released. |

## Explanation

These service calls return the resource to the semaphore specified by parameter *semid* (adds 0x1 to the semaphore counter).

If a task is queued in the wait queue of the target semaphore when this service call is issued, the counter manipulation processing is not performed but the resource is passed to the relevant task (first task of wait queue).

As a result, the relevant task is unlinked from the wait queue and is moved from the WAITING state (waiting state for a semaphore resource) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

Note 1 If the first task linked in the wait queue is moved to the READY state after this service call is issued, this service call also re-queues the task at the end of the ready queue corresponding to the priority of the task.

Note 2 The semaphore counter managed by the RI78V4 is configured in 7-bit widths. If the number of resources exceeds the maximum count value 127 as a result of issuing this service call, the counter manipulation processing is therefore not performed but "E_QOVR" is returned.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

| Macro | Value | Description |
|---|---|---|
| E_QOVR | -43 | Queue overflow (release will exceed maximum resource count "127"). |

# wai_sem

## Outline

Acquire semaphore resource (waiting forever).

## C format

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

## Assembly format

```
MOV     A, #semid
CALL    !!_wai_sem
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | ID *semid*; | ID number of the semaphore from which resource is acquired. |

## Explanation

This service call acquires a resource from the semaphore specified by parameter *semid* (subtracts 0x1 from the semaphore counter).

If a resource could not be acquired from the target semaphore (semaphore counter is set to 0x0) when this service call is issued, the counter manipulation processing is not performed but the invoking task is queued to the target semaphore wait queue in the order of resource acquisition request (FIFO order).

As a result, the invoking task is unlinked from the ready queue and is moved from the RUNNING state to the WAITING state (waiting state for a semaphore state).

| Waiting State for a Semaphore State Cancel Operation | Return Value |
|---|---|
| The resource was returned to the target semaphore as a result of issuing sig_sem. | E_OK |
| The resource was returned to the target semaphore as a result of issuing isig_sem. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_RLWAI | -49 | Forced release from waiting (accept rel_wai/irel_wai while waiting). |

# pol_sem

## Outline

Acquire semaphore resource (polling).

## C format

```
ER      pol_sem ( ID semid );
```

## Assembly format

```
MOV     A, #semid
CALL    !!_pol_sem
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID semid;` | ID number of the semaphore from which resource is acquired. |

## Explanation

This service call acquires a resource from the semaphore specified by parameter *semid* (subtracts 0x1 from the semaphore counter).

If a resource could not be acquired from the target semaphore (semaphore counter is set to 0x0) when this service call is issued, the counter manipulation processing is not performed but "E_TMOUT" is returned.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_TMOUT | -50 | Polling failure. |

# twai_sem

## Outline

Acquire semaphore resource (with timeout).

## C format

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

## Assembly format

```
MOVW    BC, #tmout_lo
MOVW    DE, #tmout_hi
MOV     A,  #semid
CALL    !!_twai_sem
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | ID *semid*; | ID number of the semaphore from which resource is acquired. |
| I | TMO *tmout*; | Specified timeout (unit: ticks).<br>TMO_FEVR: Waiting forever.<br>TMO_POL: Polling.<br>Value: Specified timeout. |

## Explanation

This service call acquires a resource from the semaphore specified by parameter *semid* (subtracts 0x1 from the semaphore counter).

If a resource could not be acquired from the target semaphore (semaphore counter is set to 0x0) when this service call is issued, the counter manipulation processing is not performed but the invoking task is queued to the target semaphore wait queue in the order of resource acquisition request (FIFO order).

As a result, the invoking task is unlinked from the ready queue and is moved from the RUNNING state to the WAITING state (waiting state for a semaphore resource).

| Waiting State for a Semaphore Resource Cancel Operation | Return Value |
|---|---|
| The resource was returned to the target semaphore as a result of issuing sig_sem. | E_OK |
| The resource was returned to the target semaphore as a result of issuing isig_sem. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |
| Polling failure or timeout. | E_TMOUT |

Note When TMO_FEVR is specified for wait time *tmout*, processing equivalent to wai_sem will be executed. When TMO_POL is specified, processing equivalent to pol_sem will be executed.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_RLWAI | -49 | Forced release from waiting (accept rel_wai/irel_wai while waiting). |
| E_TMOUT | -50 | Polling failure or timeout. |

# ref_sem

## Outline

Reference semaphore state.

## C format

```
ER      ref_sem ( ID semid, T_RSEM *pk_rsem );
```

## Assembly format

```
MOVW    BC, #LOWW(_pk_rsem)
MOV     A, #semid
CALL    !!_ref_sem
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID semid;` | ID number of the semaphore to be referenced. |
| O | `T_RSEM *pk_rsem;` | Pointer to the packet returning the semaphore state. |

## Explanation

Stores semaphore state packet (such as existence of waiting tasks) of the semaphore specified by parameter *semid* in the area specified by parameter *pk_rsem*.

Note For details about the semaphore state packet, refer to "12.5.2 Semaphore state packet".

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

## 12.9 Synchronization and Communication Functions (Eventflags)

The following lists the service calls provided by the RI78V4 as the synchronization and communication functions (eventflags).

Table 12-11 Synchronization and Communication Functions (Eventflags)

| Service Call | Function | Origin of Service Call |
|---|---|---|
| set_flg | Set eventflag. | Task, Non-task |
| iset_flg | Set eventflag. | Task, Non-task |
| clr_flg | Clear eventflag. | Task, Non-task |
| wai_flg | Wait for eventflag (waiting forever). | Task |
| pol_flg | Wait for eventflag (polling). | Task, Non-task |
| twai_flg | Wait for eventflag (with timeout). | Task |
| ref_flg | Reference eventflag state. | Task, Non-task |

# set_flg
# iset_flg

## Outline

Set eventflag.

## C format

```
ER      set_flg ( ID flgid, FLGPTN setptn );

ER      iset_flg ( ID flgid, FLGPTN setptn );
```

## Assembly format

```
MOVW    BC, #setptn
MOV     A,  #flgid
CALL    !!_set_flg

MOVW    BC, #setptn
MOV     A,  #flgid
CALL    !!_iset_flg
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID flgid;` | ID number of the eventflag to be set. |
| I | `FLGPTN setptn;` | Bit pattern to set (16 bits). |

## Explanation

These service calls set the result of ORing the bit pattern of the eventflag specified by parameter *flgid* and the bit pattern specified by parameter *setptn* as the bit pattern of the target eventflag.

If the required condition of the task queued to the target eventflag wait queue is satisfied when this service call is issued, the relevant task is unlinked from the wait queue at the same time as bit pattern setting processing.

As a result, the relevant task is moved from the WAITING state (waiting state for an eventflag) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

Note 1 If the task linked in the wait queue is moved to the READY state after this service call is issued, this service call also re-queues the task at the end of the ready queue corresponding to the priority of the task.

Note 2 If the bit pattern set to the target eventflag is B'1100 and the bit pattern specified by parameter *setptn* is B'1010 when this service call is issued, the bit pattern of the target eventflag is set to B'1110.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

# clr_flg

## Outline

Clear eventflag.

## C format

```
ER      clr_flg ( ID flgid, FLGPTN clrptn );
```

## Assembly format

```
MOVW    BC, #clrptn
MOV     A,  #flgid
CALL    !!_clr_flg
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID` *`flgid`*; | ID number of the eventflag to be cleared. |
| I | `FLGPTN` *`clrptn`*; | Bit pattern to clear (16 bits). |

## Explanation

This service call sets the result of ANDing the bit pattern set to the eventflag specified by parameter *flgid* and the bit pattern specified by parameter *clrptn* as the bit pattern of the target eventflag.

Note 1 This service call does not perform queuing of clear requests. If the bit pattern has been cleared, therefore, no processing is performed but it is not handled as an error.

Note 2 If the bit pattern set to the target eventflag is B'1100 and the bit pattern specified by parameter *clrptn* is B'1010 when this service call is issued, the bit pattern of the target eventflag is set to B'1110.

Note 3 This service call does not cancel tasks in the waiting state for an eventflag.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

# wai_flg

## Outline

Wait for eventflag (waiting forever).

## C format

```
ER      wai_flg ( ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn );
```

## Assembly format

```
MOVW    DE, #LOWW(_p_flgptn)
MOVW    BC, #waiptn
MOVW    AX, #(flgid | wfmode)
CALL    !!_wai_flg
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID flgid;` | ID number of the eventflag wait for. |
| I | `FLGPTN waiptn;` | Wait bit pattern (16 bits). |
| I | `MODE wfmode;` | Wait mode.<br>TWF_ANDW: AND waiting condition.<br>TWF_ORW: OR waiting condition. |
| O | `FLGPTN *p_flgptn;` | Bit pattern causing a task to be released from waiting. |

## Explanation

This service call checks whether the bit pattern specified by parameter *waiptn* and the bit pattern that satisfies the required condition specified by parameter *wfmode* are set to the eventflag specified by parameter *flgid*.

If a bit pattern that satisfies the required condition has been set for the target eventflag, the bit pattern of the target eventflag is stored in the area specified by parameter *p_flgptn*.

If the bit pattern of the target eventflag does not satisfy the required condition when this service call is issued, the invoking task is queued to the target eventflag wait queue.

As a result, the invoking task is unlinked from the ready queue and is moved from the RUNNING state to the WAITING state (waiting state for an eventflag).

| Waiting State for an Eventflag Cancel Operation | Return Value |
|---|---|
| A bit pattern that satisfies the required condition was set to the target eventflag as a result of issuing set_flg. | E_OK |
| A bit pattern that satisfies the required condition was set to the target eventflag as a result of issuing iset_flg. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |

| Waiting State for an Eventflag Cancel Operation | Return Value |
|---|---|
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |

The following shows the specification format of required condition *wfmode*.

- *wfmode* = TWF_ANDW
  Checks whether all of the bits to which 1 is set by parameter *waiptn* are set as the target eventflag.

- *wfmode* = TWF_ORW
  Checks which bit, among bits to which 1 is set by parameter *waiptn*, is set as the target eventflag.

Note 1 In the RI78V4, the number of tasks that can be queued to the eventflag wait queue is one. If this service call is issued for the eventflag to which a task is queued, therefore, "E_ILUSE" is returned regardless of whether or not the required condition is immediately satisfied.

Note 2 The RI78V4 performs bit pattern clear processing (0x0 setting) when the required condition of the target eventflag (TA_CLR attribute) is satisfied.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_ILUSE | -28 | Illegal service call use (there is already a task waiting for an eventflag with the TA_WSGL attribute). |
| E_RLWAI | -49 | Forced release from waiting (accept rel_wai/irel_wai while waiting). |

# pol_flg

## Outline

Wait for eventflag (polling).

## C format

```
ER      pol_flg ( ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn );
```

## Assembly format

```
MOVW    DE, #LOWW(_p_flgptn)
MOVW    BC, #waiptn
MOVW    AX, #(flgid | wfmode)
CALL    !!_pol_flg
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | ID *flgid*; | ID number of the eventflag wait for. |
| I | FLGPTN *waiptn*; | Wait bit pattern (16 bits). |
| I | MODE *wfmode*; | Wait mode.<br>TWF_ANDW: AND waiting condition.<br>TWF_ORW: OR waiting condition. |
| O | FLGPTN **p_flgptn*; | Bit pattern causing a task to be released from waiting. |

## Explanation

This service call checks whether the bit pattern specified by parameter *waipt*n and the bit pattern that satisfies the required condition specified by parameter *wfmode* are set to the eventflag specified by parameter *flgid*.

If the bit pattern that satisfies the required condition has been set to the target eventflag, the bit pattern of the target eventflag is stored in the area specified by parameter *p_flgptn*.

If the bit pattern of the target eventflag does not satisfy the required condition when this service call is issued, "E_TMOUT" is returned.

The following shows the specification format of required condition *wfmode*.

- *wfmode* = TWF_ANDW
  Checks whether all of the bits to which 1 is set by parameter *waiptn* are set as the target eventflag.

- *wfmode* = TWF_ORW
  Checks which bit, among bits to which 1 is set by parameter *waiptn*, is set as the target eventflag.

Note 1 In the RI78V4, the number of tasks that can be queued to the eventflag wait queue is one. If this service call is issued for the eventflag to which a task is queued, therefore, "E_ILUSE" is returned regardless of whether or not the required condition is immediately satisfied.

Note 2 The RI78V4 performs bit pattern clear processing (0x0 setting) when the required condition of the target eventflag (TA_CLR attribute) is satisfied.

Note 3 In the RI78V4, the number of tasks that can be queued to the eventflag wait queue is one. If this service call is issued for the eventflag to which a task is queued, therefore, "E_ILUSE" is returned regardless of whether or not the required condition is immediately satisfied.

Note 4 The RI78V4 performs bit pattern clear processing (0x0 setting) when the required condition of the target eventflag (TA_CLR attribute) is satisfied.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_ILUSE | -28 | Illegal service call use (there is already a task waiting for an eventflag with the TA_WSGL attribute). |
| E_TMOUT | -50 | Polling failure. |

# twai_flg

## Outline

Wait for eventflag (with timeout).

## C format

```
ER      twai_flg ( ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn, TMO tmout
);
```

## Assembly format

```
MOVW    AX, #tmout_hi
PUSH    AX
MOVW    AX, #tmout_lo
PUSH    AX
MOVW    DE, #LOWW(_p_flgptn)
MOVW    BC, #waiptn
MOVW    AX, #(flgid | wfmode)
CALL    !!_twai_flg
addw    sp, #0x0004
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | ID *flgid*; | ID number of the eventflag wait for. |
| I | FLGPTN *waiptn*; | Wait bit pattern (16 bits). |
| I | MODE *wfmode*; | Wait mode.<br>TWF_ANDW: AND waiting condition.<br>TWF_ORW: OR waiting condition. |
| O | FLGPTN **p_flgptn*; | Bit pattern causing a task to be released from waiting. |
| I | TMO *tmout*; | Specified timeout (unit: ticks).<br>TMO_FEVR: Waiting forever.<br>TMO_POL: Polling.<br>Value: Specified timeout. |

## Explanation

This service call checks whether the bit pattern specified by parameter *waiptn* and the bit pattern that satisfies the required condition specified by parameter *wfmode* are set to the eventflag specified by parameter *flgid*.

If the bit pattern that satisfies the required condition has been set to the target eventflag, the bit pattern of the target eventflag is stored in the area specified by parameter *p_flgptn*.

If the bit pattern of the target eventflag does not satisfy the required condition when this service call is issued, the invoking task is queued to the target eventflag wait queue.

As a result, the invoking task is unlinked from the ready queue and is moved from the RUNNING state to the WAITING state (waiting state for an eventflag).

The waiting state for an eventflag is cancelled in the following cases, and then moved to the READY state.

| Waiting State for an Eventflag Cancel Operation | Return Value |
|---|---|
| A bit pattern that satisfies the required condition was set to the target eventflag as a result of issuing set_flg. | E_OK |
| A bit pattern that satisfies the required condition was set to the target eventflag as a result of issuing iset_flg. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |
| Polling failure or timeout. | E_TMOUT |

The following shows the specification format of required condition *wfmode*.

- *wfmode* = TWF_ANDW
  Checks whether all of the bits to which 1 is set by parameter *waiptn* are set as the target eventflag.

- *wfmode* = TWF_ORW
  Checks which bit, among bits to which 1 is set by parameter *waiptn*, is set as the target eventflag.

Note 1 In the RI78V4, the number of tasks that can be queued to the eventflag wait queue is one. If this service call is issued for the eventflag to which a task is queued, therefore, "E_ILUSE" is returned regardless of whether or not the required condition is immediately satisfied.

Note 2 The RI78V4 performs bit pattern clear processing (0x0 setting) when the required condition of the target eventflag (TA_CLR attribute) is satisfied.

Note 3 When TMO_FEVR is specified for wait time tmout, processing equivalent to wai_flg will be executed. When TMO_POL is specified, processing equivalent to pol_flg will be executed.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_ILUSE | -28 | Illegal service call use (there is already a task waiting for an eventflag with the TA_WSGL attribute). |
| E_RLWAI | -49 | Forced release from waiting (accept rel_wai/irel_wai while waiting). |
| E_TMOUT | -50 | Polling failure or timeout. |

# ref_flg

## Outline

Reference eventflag state.

## C format

```
ER      ref_flg ( ID flgid, T_RFLG *pk_rflg );
```

## Assembly format

```
MOVW    BC, #LOWW(_pk_rflg)
MOV     A, #flgid
CALL    !!_ref_flg
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID flgid;` | ID number of the eventflag to be referenced. |
| O | `T_RFLG *pk_rflg;` | Pointer to the packet returning the eventflag state. |

## Explanation

Stores eventflag state packet (such as existence of waiting tasks) of the eventflag specified by parameter *flgid* in the area specified by parameter *pk_rflg*.

Note For details about the eventflag state packet, refer to "12.5.3 Eventflag state packet".

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

## 12.10 Synchronization and Communication Functions (Data queues)

The following shows the service calls provided by the RI78V4 as the synchronization and communication functions (data queues).

Table 12-12 Synchronization and Communication Functions (Data Queues)

| Service Call | Function | Origin of Service Call |
|---|---|---|
| snd_dtq | Send to data queue (waiting forever) | Task |
| psnd_dtq | Send to data queue (polling) | Task, Non-task, Initialization routine |
| ipsnd_dtq | Send to data queue (polling) | Task, Non-task, Initialization routine |
| tsnd_dtq | Send to data queue (with timeout) | Task |
| fsnd_dtq | Forced send to data queue | Task, Non-task, Initialization routine |
| ifsnd_dtq | Forced send to data queue | Task, Non-task, Initialization routine |
| rcv_dtq | Receive from data queue (waiting forever) | Task |
| prcv_dtq | Receive from data queue (polling) | Task, Non-task, Initialization routine |
| prcv_dtq | Receive from data queue (polling) | Task, Non-task, Initialization routine |
| trcv_dtq | Receive from data queue (with timeout) | Task |
| ref_dtq | Reference data queue state | Task, Non-task, Initialization routine |

# snd_dtq

## Outline

Send to data queue (waiting forever).

## C format

```
ER      snd_dtq ( ID dtqid, VP_INT data );
```

## Assembly format

```
MOVW    DE, !LOWW(_data+0x00002)
MOVW    BC, !LOWW(_data)
MOV     A,  #dtqid
CALL    !!_snd_dtq
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID dtqid;` | ID number of the data queue to which the data element is sent. |
| I | `VP_INT data;` | Data element to be sent to the data queue. |

## Explanation

This service call writes data specified by parameter *data* to the data queue area of the data queue specified by parameter *dtqid*.

If there is no available space for writing data in the data queue area of the target data queue when this service call is issued, this service call does not write data but queues the invoking task to the transmission wait queue of the target data queue and moves it from the RUNNING state to the WAITING state (data transmission wait state).

The sending WAITING state for a data queue is cancelled in the following cases, and then moved to the READY state.

| Sending WAITING State for a Data Queue Cancel Operation | Return Value |
|---|---|
| Available space was secured in the data queue area of the target data queue as a result of issuing rcv_dtq. | E_OK |
| Available space was secured in the data queue area of the target data queue as a result of issuing prcv_dtq. | E_OK |
| Available space was secured in the data queue area of the target data queue as a result of issuing trcv_dtq. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |

If a task has been queued to the reception wait queue of the target data queue when this service call is issued, this service call does not write data but transfers the data to the task. As a result, the task is unlinked from the reception wait queue and moves from the WAITING state (data reception wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

Note 1 Data is written to the data queue area of the target data queue in the order of the data transmission request.

Note 2 Invoking tasks are queued to the transmission wait queue of the target data queue in the order defined during configuration (FIFO order).

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_RLWAI | -49 | Forced release from the WAITING state.<br>- Accept rel_wai/irel_wai while waiting. |

# psnd_dtq
# ipsnd_dtq

## Outline

Send to data queue (polling).

## C format

```
ER      psnd_dtq ( ID dtqid, VP_INT data );
ER      ipsnd_dtq ( ID dtqid, VP_INT data );
```

## Assembly format

```
MOVW    DE, !LOWW(_data+0x00002)
MOVW    BC, !LOWW(_data)
MOV     A,  #dtqid
CALL    !!_psnd_dtq

MOVW    DE, !LOWW(_data+0x00002)
MOVW    BC, !LOWW(_data)
MOV     A,  #dtqid
CALL    !!_ipsnd_dtq
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID` *`dtqid`*; | ID number of the data queue to which the data element is sent. |
| I | `VP_INT` *`data`*; | Data element to be sent to the data queue. |

## Explanation

These service calls write data specified by parameter *data* to the data queue area of the data queue specified by parameter *dtqid*.

If there is no available space for writing data in the data queue area of the target data queue when either of these service calls is issued, data is not written but E_TMOUT is returned.

If a task has been queued to the reception wait queue of the target data queue when this service call is issued, this service call does not write data but transfers the data to the task. As a result, the task is unlinked from the reception wait queue and moves from the WAITING state (data reception wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

Note Data is written to the data queue area of the target data queue in the order of the data transmission request.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_TMOUT | -50 | Polling failure.<br>- There is no space in the target data queue. |

# tsnd_dtq

## Outline

Send to data queue (with timeout).

## C format

```
ER      tsnd_dtq ( ID dtqid, VP_INT data, TMO tmout );
```

## Assembly format

```
MOVW    DE, !LOWW(_data+0x00002)
MOVW    BC, !LOWW(_data)
MOVW    AX, #tmout_hi
PUSH    AX
MOVW    AX, #tmout_lo
PUSH    AX
MOV     A,  #dtqid
CALL    !!_tsnd_dtq
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID` *`dtqid;`* | ID number of the data queue to which the data element is sent. |
| I | `VP_INT` *`data;`* | Data element to be sent to the data queue. |
| I | `TMO` *`tmout;`* | Specified timeout (in tick).<br>TMO_FEVR: Waiting forever.<br>TMO_POL: Polling.<br>Value: Specified timeout. |

## Explanation

This service call writes data specified by parameter *data* to the data queue area of the data queue specified by parameter *dtqid*.

If there is no available space for writing data in the data queue area of the target data queue when this service call is issued, the service call does not write data but queues the invoking task to the transmission wait queue of the target data queue and moves it from the RUNNING state to the WAITING state with time (data transmission wait state).

The sending WAITING state for a data queue is cancelled in the following cases, and then moved to the READY state.

| Sending WAITING State for a Data Queue Cancel Operation | Return Value |
|---|---|
| An available space was secured in the data queue area of the target data queue as a result of issuing rcv_dtq. | E_OK |
| An available space was secured in the data queue area of the target data queue as a result of issuing prcv_dtq. | E_OK |

| Sending WAITING State for a Data Queue Cancel Operation | Return Value |
|---|---|
| An available space was secured in the data queue area of the target data queue as a result of issuing trcv_dtq. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |
| Polling failure or timeout. | E_TMOUT |

If a task has been queued to the reception wait queue of the target data queue when this service call is issued, this service call does not write data but transfers the data to the task. As a result, the task is unlinked from the reception wait queue and moves from the WAITING state (data reception wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

Note 1 Data is written to the data queue area of the target data queue in the order of the data transmission request.

Note 2 Invoking tasks are queued to the transmission wait queue of the target data queue in the order defined during configuration (FIFO order).

Note 3 TMO_FEVR is specified for wait time *tmout*, processing equivalent to snd_dtq will be executed. When TMO_POL is specified, processing equivalent to psnd_dtq /ipsnd_dtq will be executed.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_RLWAI | -49 | Forced release from the WAITING state.<br>- Accept rel_wai/irel_wai while waiting. |
| E_TMOUT | -50 | Timeout.<br>- Polling failure or timeout. |

# fsnd_dtq
# ifsnd_dtq

## Outline

Forced send to data queue.

## C format

```
ER      fsnd_dtq ( ID dtqid, VP_INT data );
```

## Assembly format

```
MOVW    DE, !LOWW(_data+0x00002)
MOVW    BC, !LOWW(_data)
MOV     A,  #dtqid
CALL    !!_fsnd_dtq

MOVW    DE, !LOWW(_data+0x00002)
MOVW    BC, !LOWW(_data)
MOV     A,  #dtqid
CALL    !!_ifsnd_dtq
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | ID *dtqid*; | ID number of the data queue to which the data element is sent. |
| I | VP_INT *data*; | Data element to be sent to the data queue. |

## Explanation

These service calls write data specified by parameter *data* to the data queue area of the data queue specified by parameter *dtqid*.

If there is no available space for writing data in the data queue area of the target data queue when either of these service calls is issued, the service call overwrites data to the area with the oldest data that was written.

If a task has been queued to the reception wait queue of the target data queue when this service call is issued, this service call does not write data but transfers the data to the task. As a result, the task is unlinked from the reception wait queue and moves from the WAITING state (data reception wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

# rcv_dtq

## Outline

Receive from data queue (waiting forever).

## C format

```
ER      rcv_dtq ( ID dtqid, VP_INT *p_data );
```

## Assembly format

```
MOVW    BC, #LOWW(_data)
MOV     A,  #dtqid
CALL    !!_rcv_dtq
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID dtqid;` | ID number of the data queue from which a data element is received. |
| O | `VP_INT *p_data;` | Data element received from the data queue. |

## Explanation

This service call reads data in the data queue area of the data queue specified by parameter *dtqid* and stores it to the area specified by parameter *p_data*.

If no data could be read from the data queue area of the target data queue (no data has been written to the data queue area) when this service call is issued, the service call does not read data but queues the invoking task to the reception wait queue of the target data queue and moves it from the RUNNING state to the WAITING state (data reception wait state).

The receiving WAITING state for a data queue is cancelled in the following cases, and then moved to the READY state.

| Receiving WAITING State for a Data Queue Cancel Operation | Return Value |
|---|---|
| Data was written to the data queue area of the target data queue as a result of issuing snd_dtq. | E_OK |
| Data was written to the data queue area of the target data queue as a result of issuing psnd_dtq. | E_OK |
| Data was written to the data queue area of the target data queue as a result of issuing ipsnd_dtq. | E_OK |
| Data was written to the data queue area of the target data queue as a result of issuing tsnd_dtq. | E_OK |
| Data was written to the data queue area of the target data queue as a result of issuing tsnd_dtq. | E_OK |
| Data was written to the data queue area of the target data queue as a result of issuing ifsnd_dtq. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |

Note 1 Invoking tasks are queued to the reception wait queue of the target data queue in the order of the data reception request.

Note 2 If the receiving for a data queue is forcibly released by issuing rel_wai or irel_wai, the contents of the area specified by parameter *p_data* will be undefined.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_RLWAI | -49 | Forced release from the WAITING state.<br>- Accept rel_wai/irel_wai while waiting. |

# prcv_dtq

## Outline

Receive from data queue (polling).

## C format

```
ER      prcv_dtq ( ID dtqid, VP_INT *p_data );
```

## Assembly format

```
MOVW    BC, #LOWW(_data)
MOV     A,  #dtqid
CALL    !!_prcv_dtq
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID dtqid;` | ID number of the data queue from which a data element is received. |
| O | `VP_INT *p_data;` | Data element received from the data queue. |

## Explanation(s)

These service calls read data in the data queue area of the data queue specified by parameter *dtqid* and stores it to the area specified by parameter *p_data*.

If no data could be read from the data queue area of the target data queue (no data has been written to the data queue area) when either of these service calls is issued, the service call does not read data but E_TMOUT is returned.

Note  If no data could be read from the data queue area of the target data queue (no data has been written to the data queue area) when either of these service calls is issued, the contents in the area specified by parameter *p_data* become undefined.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_TMOUT | -50 | Polling failure.<br>- No data exists in the target data queue. |

# trcv_dtq

## Outline

Receive from data queue (with timeout).

## C format

```
ER      trcv_dtq ( ID dtqid, VP_INT *p_data, TMO tmout );
```

## Assembly format

```
MOVW    AX, #tmout_hi
PUSH    AX
MOVW    AX, #tmout_lo
PUSH    AX
MOVW    BC, #LOWW(_data)
MOV     A,  #dtqid
CALL    !!_trcv_dtq
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID dtqid;` | ID number of the data queue from which a data element is received. |
| O | `VP_INT *p_data;` | Data element received from the data queue. |
| I | `TMO tmout;` | Specified timeout (in tick).<br>TMO_FEVR: Waiting forever.<br>TMO_POL: Polling.<br>Value: Specified timeout. |

## Explanation

This service call reads data in the data queue area of the data queue specified by parameter *dtqid* and stores it to the area specified by parameter *p_data*.

If no data could be read from the data queue area of the target data queue (no data has been written to the data queue area) when this service call is issued, the service call does not read data but queues the invoking task to the reception wait queue of the target data queue and moves it from the RUNNING state to the WAITING state with time out (data reception wait state).

The receiving WAITING state for a data queue is cancelled in the following cases, and then moved to the READY state.

| Receiving WAITING State for a Data Queue Cancel Operation | Return Value |
|---|---|
| Data was written to the data queue area of the target data queue as a result of issuing snd_dtq. | E_OK |
| Data was written to the data queue area of the target data queue as a result of issuing psnd_dtq. | E_OK |

| Receiving WAITING State for a Data Queue Cancel Operation | Return Value |
|---|---|
| Data was written to the data queue area of the target data queue as a result of issuing ipsnd_dtq. | E_OK |
| Data was written to the data queue area of the target data queue as a result of issuing tsnd_dtq. | E_OK |
| Data was written to the data queue area of the target data queue as a result of issuing fsnd_dtq. | E_OK |
| Data was written to the data queue area of the target data queue as a result of issuing ifsnd_dtq. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |
| Polling failure or timeout. | E_TMOUT |

Note 1 Invoking tasks are queued to the reception wait queue of the target data queue in the order of the data reception request.

Note 2 If the data reception wait state is cancelled because irel_wai or irel_wai was issued or the wait time elapsed, the contents in the area specified by parameter *p_data* become undefined.

Note 3 TMO_FEVR is specified for wait time *tmout*, processing equivalent to rcv_dtq will be executed. When TMO_POL is specified, processing equivalent to prcv_dtq will be executed.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_RLWAI | -49 | Forced release from the WAITING state.<br>- Accept rel_wai/irel_wai while waiting. |
| E_TMOUT | -50 | Timeout.<br>- Polling failure or timeout. |

# ref_dtq

## Outline

Reference data queue state.

## C format

```
ER      ref_dtq ( ID dtqid, T_RDTQ *pk_rdtq );
```

## Assembly format

```
MOVW    BC, #LOWW(_pk_rdtq)
MOV     A, #dtqid
CALL    !!_ref_dtq
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID dtqid;` | ID number of the data queue to be referenced. |
| O | `T_RDTQ *pk_rdtq;` | Pointer to the packet returning the data queue state. |

[Data queue state packet: T_RDTQ]

```
typedef struct  t_rdtq {
    ID      stskid;         /*Existence of tasks waiting for data transmission*/
    ID      rtskid;         /*Existence of tasks waiting for data reception*/
    UINT    sdtqcnt;        /*Number of data elements in data queue*/
} T_RDTQ;
```

## Explanation

These service calls store the detailed information of the data queue (existence of waiting tasks, number of data elements in the data queue, etc.) specified by parameter *dtqid* into the area specified by parameter *pk_rdtq*.

Note For details about the data queue state packet, refer to "12.5.4 Data queue state packet".

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

## 12.11 Synchronization and Communication Functions (Mailboxes)

The following lists the service calls provided by the RI78V4 as the synchronization and communication functions (mailboxes).

Table 12-13 Synchronization and Communication Functions (Mailboxes)

| Service Call | Function | Origin of Service Call |
|---|---|---|
| snd_mbx | Send to mailbox. | Task, Non-task |
| rcv_mbx | Receive from mailbox (waiting forever). | Task |
| prcv_mbx | Receive from mailbox (polling). | Task, Non-task |
| trcv_mbx | Receive from mailbox (with timeout). | Task |
| ref_mbx | Reference mailbox state. | Task, Non-task |

# snd_mbx

## Outline

Send to mailbox.

## C format

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

## Assembly format

```
SUBW    SP, #0x06
MOVW    [SP+0x02], AX
MOVW    AX, BC
MOVW    [SP+0x04], AX
MOVW    AX, SP
MOVW    BC, AX
MOV     A, #mbxid
CALL    !!_snd_mbx
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | ID *mbxid*; | ID number of the mailbox to which the message is sent. |
| I | T_MSG **pk_msg*; | Start address of the message packet to be sent to the mailbox. |

## Explanation

This service call transmits the message specified by parameter *pk_msg* to the mailbox specified by parameter *mbxid* (queues the message in the wait queue).

If a task is queued to the target mailbox wait queue when this service call is issued, the message is not queued but handed over to the relevant task (first task of the wait queue).

As a result, the relevant task is unlinked from the wait queue and is moved from the WAITING state (receiving waiting for a mailbox) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

Note 1 If the first task of the wait queue is moved to the READY state after this service call is issued, this service call also re-queues the task at the end of the ready queue corresponding to the priority of the task.

Note 2 Messages are queued to the target mailbox wait queue in the order defined by Attribute (queuing method): mbxatr during configuration (FIFO order or priority order).

Note 3 With the RI78V4 mailbox, only the start address of the message is handed over to the receiving processing program, but the message contents are not copied to a separate area. The message contents can therefore be rewritten even after this service call is issued.

Note 4 For details about the message packet, refer to "12.5.5 Message packet".

### Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

# rcv_mbx

## Outline

Receive from mailbox (waiting forever).

## C format

```
ER      rcv_mbx ( ID mbxid, T_MSG **ppk_msg );
```

## Assembly format

```
MOVW    BC, #LOWW(_ppk_msg)
MOV     A, #mbxid
CALL    !!_rcv_mbx
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | ID *mbxid*; | ID number of the mailbox from which a message is received. |
| O | T_MSG ***ppk_msg*; | Start address of the message packet received from the mailbox. |

## Explanation

This service call receives a message from the mailbox specified by parameter *mbxid*, and stores its start address in the area specified by parameter *ppk_msg*.

If the message could not be received from the target mailbox (no messages were queued in the wait queue) when this service call is issued, message reception processing is not executed but the invoking task is queued to the target mailbox wait queue in the order of message reception request (FIFO order).

As a result, the invoking task is unlinked from the ready queue and is moved from the RUNNING state to the WAITING state (receiving waiting state for a mailbox).

| Receiving Waiting State for a mailbox Cancel Operation | Return Value |
|---|---|
| A message was transmitted to the target mailbox as a result of issuing snd_mbx. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |

Note For details about the message packet, refer to "12.5.5 Message packet".

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_RLWAI | -49 | Forced release from waiting (accept rel_wai/irel_wai while waiting). |

# prcv_mbx

## Outline

Receive from mailbox (polling).

## C format

```
ER      prcv_mbx ( ID mbxid, T_MSG **ppk_msg );
```

## Assembly format

```
MOVW    BC, #LOWW(_ppk_msg)
MOV     A, #mbxid
CALL    !!_prcv_mbx
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID` *`mbxid;`* | ID number of the mailbox from which a message is received. |
| O | `T_MSG` *`**ppk_msg;`* | Start address of the message packet received from the mailbox. |

## Explanation

This service call receives a message from the mailbox specified by parameter *mbxid*, and stores its start address in the area specified by parameter *ppk_msg*.

If the message could not be received from the target mailbox (no messages were queued in the wait queue) when this service call is issued, message reception processing is not executed but "E_TMOUT" is returned.

Note For details about the message packet, refer to "12.5.5 Message packet".

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_TMOUT | -50 | Polling failure. |

# trcv_mbx

## Outline

Receive from mailbox (with timeout).

## C format

```
ER      trcv_mbx ( ID mbxid, T_MSG **ppk_msg, TMO tmout );
```

## Assembly format

```
MOVW    AX, #tmout_hi
PUSH    AX
MOVW    AX, #tmout_lo
PUSH    AX
MOVW    BC, #LOWW(_ppk_msg)
MOV     A, #mbxid
CALL    !!_trcv_mbx
ADDW    SP, #0x0004
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID` *mbxid*; | ID number of the mailbox from which a message is received. |
| O | `T_MSG` `**`*ppk_msg*; | Start address of the message packet received from the mailbox. |
| I | `TMO` *tmout*; | Specified timeout (unit: ticks).<br>TMO_FEVR: Waiting forever.<br>TMO_POL: Polling.<br>Value: Specified timeout. |

## Explanation

This service call receives a message from the mailbox specified by parameter *mbxid*, and stores its start address in the area specified by parameter *ppk_msg*.

If the message could not be received from the target mailbox (no messages were queued in the wait queue) when this service call is issued, message reception processing is not executed but the invoking task is queued to the target mailbox wait queue in the order of message reception request (FIFO order).

As a result, the invoking task is unlinked from the ready queue and is moved from the RUNNING state to the WAITING state (receiving waiting state for a mailbox).

The receiving waiting state for a mailbox is cancelled in the following cases, and then moved to the READY state.

| Receiving Waiting State for a mailbox Cancel Operation | Return Value |
|---|---|
| A message was transmitted to the target mailbox as a result of issuing snd_mbx. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |

| Receiving Waiting State for a mailbox Cancel Operation | Return Value |
|---|---|
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |
| Polling failure or timeout. | E_TMOUT |

Note 1 When TMO_FEVR is specified for wait time *tmout*, processing equivalent to rcv_mbx will be executed. When TMO_POL is specified, processing equivalent to prcv_mbx will be executed.

Note 2 For details about the message packet, refer to "12.5.5 Message packet".

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_RLWAI | -49 | Forced release from waiting (accept rel_wai/irel_wai while waiting). |
| E_TMOUT | -50 | Polling failure or timeout. |

# ref_mbx

## Outline

Reference mailbox state.

## C format

```
ER      ref_mbx ( ID mbxid, T_RMBX *pk_rmbx );
```

## Assembly format

```
MOVW    BC, #LOWW(_pk_rmbx)
MOV     A,  #mbxid
CALL    !!_ref_mbx
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID mbxid;` | ID number of the mailbox to be referenced. |
| O | `T_RMBX *pk_rmbx;` | Pointer to the packet returning the mailbox state. |

## Explanation

Stores mailbox state packet (such as existence of waiting tasks) of the mailbox specified by parameter *mbxid* in the area specified by parameter *pk_rmbx*.

Note For details about the mailbox state packet, refer to "12.5.6 Mailbox state packet".

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

## 12.12 Memory Pool Management Functions

The following lists the service calls provided by the RI78V4 as the memory pool management functions.

Table 12-14 Memory Pool Management Functions

| Service Call | Function | Origin of Service Call |
|---|---|---|
| get_mpf | Acquire fixed-sized memory block (waiting forever). | Task |
| pget_mpf | Acquire fixed-sized memory block (polling). | Task, Non-task |
| tget_mpf | Acquire fixed-sized memory block (with timeout). | Task |
| rel_mpf | Release fixed-sized memory block. | Task, Non-task |
| ref_mpf | Reference fixed-sized memory pool state. | Task, Non-task |

# get_mpf

## Outline

Acquire fixed-sized memory block (waiting forever).

## C format

```
ER      get_mpf ( ID mpfid, VP *p_blk );
```

## Assembly format

```
MOVW    BC, #LOWW(_p_blk)
MOV     A,  #mpfid
CALL    !!_get_mpf
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | ID *mpfid;* | ID number of the fixed-sized memory pool from which a memory block is acquired. |
| O | VP **p_blk;* | Start address of the acquired memory block. |

## Explanation

This service call acquires the memory block from the fixed-sized memory pool specified by parameter *mpfid* and stores the start address in the area specified by parameter *p_blk*.

If a memory block could not be acquired from the target fixed-sized memory pool (no available memory blocks exist) when this service call is issued, memory block acquisition processing is not performed but the invoking task is queued to the target fixed-sized memory pool wait queue in the order of memory block acquisition request (FIFO order).

As a result, the invoking task is unlinked from the ready queue and is moved from the RUNNING state to the WAITING state (waiting state for a fixed-sized memory block).

The waiting state for a fixed-sized memory block is cancelled in the following cases, and then moved to the READY state.

| Waiting State for a Fixed-sized Memory Block Cancel Operation | Return Value |
|---|---|
| A memory block was returned to the target fixed-sized memory pool as a result of issuing rel_mpf. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_RLWAI | -49 | Forced release from waiting (accept rel_wai/irel_wai while waiting). |

# pget_mpf

## Outline

Acquire fixed-sized memory block (polling).

## C format

```
ER      pget_mpf ( ID mpfid, VP *p_blk );
```

## Assembly format

```
MOVW    BC, #LOWW(_p_blk)
MOV     A, #mpfid
CALL    !!_pget_mpf
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID` *`mpfid;`* | ID number of the fixed-sized memory pool from which a memory block is acquired. |
| O | `VP` *`*p_blk;`* | Start address of the acquired memory block. |

## Explanation

This service call acquires the memory block from the fixed-sized memory pool specified by parameter *mpfid* and stores the start address in the area specified by parameter *p_blk*.

If a memory block could not be acquired from the target fixed-sized memory pool (no available memory blocks exist) when this service call is issued, memory block acquisition processing is not performed but "E_TMOUT" is returned.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_TMOUT | -50 | Polling failure. |

# tget_mpf

## Outline

Acquire fixed-sized memory block (with timeout).

## C format

```
ER      tget_mpf ( ID mpfid, VP *p_blk, TMO tmout );
```

## Assembly format

```
MOVW    AX, #tmout_hi
PUSH    AX
MOVW    AX, #tmout_lo
PUSH    AX
MOVW    BC, #LOWW(_p_blk)
MOV     A, #mpfid
CALL    !!_tget_mpf
ADDW    SP, #0x0004
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID mpfid;` | ID number of the fixed-sized memory pool from which a memory block is acquired. |
| O | `VP *p_blk;` | Start address of the acquired memory block. |
| I | `TMO tmout;` | Specified timeout (unit: ticks).<br>TMO_FEVR: Waiting forever.<br>TMO_POL: Polling.<br>Value: Specified timeout. |

## Explanation

This service call acquires the memory block from the fixed-sized memory pool specified by parameter *mpfid* and stores the start address in the area specified by parameter *p_blk*.

If a memory block could not be acquired from the target fixed-sized memory pool (no available memory blocks exist) when this service call is issued, memory block acquisition processing is not performed but the invoking task is queued to the target fixed-sized memory pool wait queue in the order of memory block acquisition request (FIFO order).

As a result, the invoking task is unlinked from the ready queue and is moved from the RUNNING state to the WAITING state (waiting state for a fixed-sized memory block).

The waiting state for a fixed-sized memory block is cancelled in the following cases, and then moved to the READY state.

| Waiting State for a Fixed-sized Memory Block Cancel Operation | Return Value |
|---|---|
| A memory block was returned to the target fixed-sized memory pool as a result of issuing rel_mpf. | E_OK |
| Forced release from waiting (accept rel_wai while waiting). | E_RLWAI |
| Forced release from waiting (accept irel_wai while waiting). | E_RLWAI |
| Polling failure or timeout. | E_TMOUT |

Note When TMO_FEVR is specified for wait time *tmout*, processing equivalent to get_mpf will be executed. When TMO_POL is specified, processing equivalent to pget_mpf will be executed.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |
| E_RLWAI | -49 | Forced release from waiting (accept rel_wai/irel_wai while waiting). |
| E_TMOUT | -50 | Polling failure or timeout. |

# rel_mpf

## Outline

Release fixed-sized memory block.

## C format

```
ER      rel_mpf ( ID mpfid, VP blk );
```

## Assembly format

```
MOVW    BC, #LOWW(_blk)
MOV     A, #mpfid
CALL    !!_rel_mpf
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID` *`mpfid;`* | ID number of the fixed-sized memory pool to which the memory block is released. |
| I | `VP` *`blk;`* | Start address of the memory block to be released. |

## Explanation

This service call returns the memory block specified by parameter *blk* to the fixed-sized memory pool specified by parameter *mpfid*.

If a task is queued to the target fixed-sized memory pool wait queue when this service call is issued, memory block return processing is not performed but memory blocks are returned to the relevant task (first task of wait queue).

As a result, the relevant task is unlinked from the wait queue and is moved from the WAITING state (waiting state for a fixed-sized memory block) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

Note 1 If the first task of the wait queue is moved to the READY state after this service call is issued, this service call also re-queues the task at the end of the ready queue corresponding to the priority of the task.

Note 2 The RI78V4 does not clear the memory blocks before returning them. The contents of the returned memory blocks are therefore undefined.

### Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

# ref_mpf

## Outline

Reference fixed-sized memory pool state.

## C format

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

## Assembly format

```
MOVW    BC, #LOWW(_pk_rmpf)
MOV     A,  #mpfid
CALL    !!_ref_mpf
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID mpfid;` | ID number of the fixed-sized memory pool to be referenced. |
| O | `T_RMPF *pk_rmpf;` | Pointer to the packet returning the fixed-sized memory pool state. |

## Explanation

Stores fixed-sized memory pool state packet (such as existence of waiting tasks) of the fixed-sized memory pool specified by parameter *mpfid* in the area specified by parameter *pk_rmpf*.

Note For details about the fixed-sized memory pool state packet, refer to "12.5.7 Fixed-sized memory pool state packet".

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

## 12.13 Time Management Functions

The following lists the service calls provided by the RI78V4 as the time management functions.

Table 12-15 Time Management Functions

| Service Call | Function | Origin of Service Call |
|---|---|---|
| sta_cyc | Start cyclic handler operation. | Task, Non-task |
| stp_cyc | Stop cyclic handler operation. | Task, Non-task |
| ref_cyc | Reference cyclic handler state. | Task, Non-task |

# sta_cyc

## Outline

Start cyclic handler operation.

## C format

```
ER      sta_cyc ( ID cycid );
```

## Assembly format

```
MOV     A, #cycid
CALL    !!_sta_cyc
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | ID *cycid*; | ID number of the cyclic handler operation to be started. |

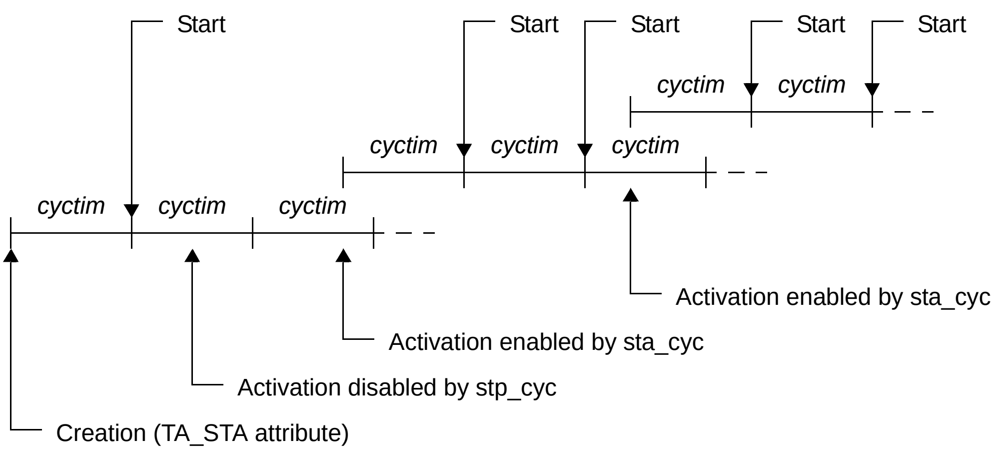
## Explanation

This service call moves the cyclic handler specified by parameter *cycid* from the non-operational state (STP state) to operational state (STA state).

As a result, the target cyclic handler is handled as an activation target of the RI78V4.

Note This service call does not perform queuing of start requests. If the target cyclic handler has been moved to the operational state (STA state), only activation cycle re-set processing is executed.
The relative time interval from the output of this service call until the first activation request is output is always the activation phase (activation cycle cyctim) using the output of this service call as the reference point.

[ Cyclic handler activation image ]



### Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

# stp_cyc

## Outline

Stop cyclic handler operation.

## C format

```
ER      stp_cyc ( ID cycid );
```

## Assembly format

```
MOV     A, #cycid
CALL    !!_stp_cyc
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID cycid;` | ID number of the cyclic handler operation to be stopped. |

## Explanation

This service call moves the cyclic handler specified by parameter *cycid* from the operational state (STA state) to non-operational state (STP state).
As a result, the target cyclic handler is excluded from activation targets of the RI78V4 until issuance of sta_cyc.

Note  This service call does not perform queuing of stop requests. If the target cyclic handler has been moved to the non-operational state (STP state), therefore, no processing is performed but it is not handled as an error.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

# ref_cyc

## Outline

Reference cyclic handler state.

## C format

```
ER      ref_cyc ( ID cycid, T_RCYC *pk_rcyc );
```

## Assembly format

```
MOVW    BC, #LOWW(_pk_rcyc)
MOV     A,  #cycid
CALL    !!_ref_cyc
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | `ID cycid;` | ID number of the cyclic handler to be referenced. |
| O | `T_RCYC *pk_rcyc;` | Pointer to the packet returning the cyclic handler state. |

## Explanation

Stores cyclic handler state packet (such as current status) of the cyclic handler specified by parameter *cycid* in the area specified by parameter *pk_rcyc*.

Note For details about the cyclic handler state packet, refer to "12.5.8 Cyclic handler state packet".

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

## 12.14 System State Management Functions

The following lists the service calls provided by the RI78V4 as the system state management functions.

Table 12-16 System State Management Functions

| Service Call | Function | Origin of Service Call |
| --- | --- | --- |
| rot_rdq | Rotate task precedence. | Task, Non-task |
| irot_rdq | Rotate task precedence. | Task, Non-task |
| get_tid | Reference task ID in the RUNNING state. | Task, Non-task |
| iget_tid | Reference task ID in the RUNNING state. | Task, Non-task |
| loc_cpu | Lock the CPU. | Task, Non-task |
| iloc_cpu | Lock the CPU. | Task, Non-task |
| unl_cpu | Unlock the CPU. | Task, Non-task |
| iunl_cpu | Unlock the CPU. | Task, Non-task |
| dis_dsp | Disable dispatching. | Task |
| ena_dsp | Enable dispatching. | Task |
| sns_ctx | Reference contexts. | Task, Non-task |
| sns_loc | Reference CPU state. | Task, Non-task |
| sns_dsp | Reference dispatching state. | Task, Non-task |
| sns_dpn | Reference dispatch pending state. | Task, Non-task |

# rot_rdq
# irot_rdq

## Outline

Rotate task precedence.

## C format

```
ER      rot_rdq ( PRI tskpri );

ER      irot_rdq ( PRI tskpri );
```

## Assembly format

```
MOV     A, #tskpri
CALL    !!_rot_rdq

MOV     A, #tskpri
CALL    !!_irot_rdq
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| I | PRI *tskpri;* | Priority of the tasks whose precedence is rotated.<br>TPRI_SELF: Current priority of the invoking task.<br>Value: Priority of the tasks whose precedence is rotated. |

## Explanation

This service call re-queues the first task of the ready queue corresponding to the priority specified by parameter *tskpri* to the end of the queue to change the task execution order explicitly.

Note 1 This service call does not perform queuing of rotation requests. If no task is queued to the ready queue corresponding to the relevant priority, therefore, no processing is performed but it is not handled as an error.

Note 2 Round-robin scheduling can be implemented by issuing this service call via a cyclic handler in a constant cycle.

Note 3 The ready queue is a hash table that uses priority as the key, and tasks that have entered an executable state (READY state or RUNNING state) are queued in FIFO order.
Therefore, the scheduler realizes the RI78V4's Scheduling System by executing task detection processing from the highest priority level of the ready queue upon activation, and upon detection of queued tasks, giving the CPU use right to the first task of the proper priority level.

### Return value

| Macro | Value | Description |
| --- | --- | --- |
| E_OK | 0 | Normal completion. |

# get_tid
# iget_tid

## Outline

Reference task ID in the RUNNING state.

## C format

```
ER      get_tid ( ID *p_tskid );

ER      iget_tid ( ID *p_tskid );
```

## Assembly format

```
SUBW    SP, #0x06
MOVW    [SP+0x02], AX
MOVW    AX, BC
MOVW    [SP+0x04], AX
MOVW    AX, SP
CALL    !!_get_tid

SUBW    SP, #0x06
MOVW    [SP+0x02], AX
MOVW    AX, BC
MOVW    [SP+0x04], AX
MOVW    AX, SP
CALL    !!_iget_tid
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| O | ID *p_tskid; | ID number of the task in the RUNNING state. |

## Explanation

These service calls store the ID of a task in the RUNNING state in the area specified by parameter *p_tskid*.

Note This service call stores TSK_NONE in the area specified by parameter *p_tskid* if no tasks that have entered the RUNNING state exist (all tasks in the IDLE state).

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

# loc_cpu
# iloc_cpu

## Outline

Lock the CPU.

## C format

```
ER      loc_cpu ( void );

ER      iloc_cpu ( void );
```

## Assembly format

```
CALL    !!_loc_cpu

CALL    !!_iloc_cpu
```

## Parameter(s)

None.

## Explanation

These service calls change the system status type to the CPU locked state.

As a result, maskable interrupt acknowledgment processing is prohibited during the interval from this service call is issued until unl_cpu or iunl_cpu is issued, and service call issuance is also restricted.

If a maskable interrupt is created during the interval from this service call is issued until unl_cpu or iunl_cpu is issued, the RI78V4 delays transition to the relevant interrupt processing (interrupt handler) until either unl_cpu or iunl_cpu is issued.

The service calls that can be issued in the CPU locked state are limited to the one listed below.

| Service Call | Function |
|---|---|
| loc_cpu, iloc_cpu | Lock the CPU. |
| unl_cpu, iunl_cpu | Unlock the CPU. |
| sns_ctx | Reference contexts. |
| sns_loc | Reference CPU state. |
| sns_dsp | Reference dispatching state. |
| sns_dpn | Reference dispatch pending state. |

Note 1 The CPU locked state changed by issuing this service call must be cancelled before the processing program that issued this service call ends.

Note 2 This service call does not perform queuing of lock requests. If the system is in the CPU locked state, therefore, no processing is performed but it is not handled as an error.

Note 3 The RI78V4 implements disabling of maskable interrupt acknowledgment bu manipulating the interrupt mask flag register (MK*xx*) and the in-service priority flag (ISP*x*) of the program status word (PSW). Therefore, manipulating of these registers from the processing program is prohibited from when this service call is issued until unl_cpu or iunl_cpu is issued.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

# unl_cpu
# iunl_cpu

## Outline

Unlock the CPU.

## C format

```
ER      unl_cpu ( void );

ER      iunl_cpu ( void );
```

## Assembly format

```
CALL    !!_unl_cpu

CALL    !!_iunl_cpu
```

## Parameter(s)

None.

## Explanation

These service calls change the system status to the CPU unlocked state.

As a result, acknowledge processing of maskable interrupts prohibited through issuance of either loc_cpu or iloc_cpu is enabled, and the restriction on service call issuance is released.

If a maskable interrupt is created during the interval from when either loc_cpu or iloc_cpu is issued until this service call is issued, the RI78V4 delays transition to the relevant interrupt processing (interrupt handler) until this service call is issued.

Note 1 This service call does not perform queuing of cancellation requests. If the system is in the CPU unlocked state, therefore, no processing is performed but it is not handled as an error.

Note 2 The RI78V4 implements enabling of maskable interrupt acknowledgment bu manipulating the interrupt mask flag register (MK*xx*) and the in-service priority flag (ISP*x*) of the program status word (PSW). Therefore, manipulating of these registers from the processing program is prohibited from when loc_cpu or iloc_cpu is issued until this service call is issued.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

# dis_dsp

## Outline

Disable dispatching.

## C format

```
ER      dis_dsp ( void );
```

## Assembly format

```
CALL    !!_dis_dsp
```

## Parameter(s)

None.

## Explanation

This service call changes the system status to the dispatching disabled state.

As a result, dispatch processing (task scheduling) is disabled from when this service call is issued until ena_dsp is issued.

If a service call (chg_pri, sig_sem, etc.) accompanying dispatch processing is issued during the interval from when this service call is issued until ena_dsp is issued, the RI78V4 executes only processing such as queue manipulation, counter manipulation, etc., and the actual dispatch processing is delayed until ena_dsp is issued, upon which the actual dispatch processing is performed in batch.

Note 1 This service call does not perform queuing of disable requests. If the system is in the dispatching disabled state, therefore, no processing is performed but it is not handled as an error.

Note 2 The dispatching disabled state changed by issuing this service call must be cancelled before the task that issued this service call moves to the DORMANT state.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

# ena_dsp

## Outline

Enable dispatching.

## C format

```
ER      ena_dsp ( void );
```

## Assembly format

```
CALL    !!_ena_dsp
```

## Parameter(s)

None.

## Explanation

This service call changes the system status to the dispatching enabled state.

As a result, dispatch processing (task scheduling) that has been disabled by issuing dis_dsp is enabled.

If a service call (chg_pri, sig_sem, etc.) accompanying dispatch processing is issued during the interval from when dis_dsp is issued until this service call is issued, the RI78V4 executes only processing such as queue manipulation, counter manipulation, etc., and the actual dispatch processing is delayed until this service call is issued, upon which the actual dispatch processing is performed in batch.

Note  This service call does not perform queuing of enable requests. If the system is in the dispatching enabled state, therefore, no processing is performed but it is not handled as an error.

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

# sns_ctx

## Outline

Reference contexts.

## C format

```
BOOL    sns_ctx ( void );
```

## Assembly format

```
CALL    !!_sns_ctx
```

## Parameter(s)

None.

## Explanation

This service call acquires the context type of the processing program that issued this service call (non-task context or task context).

When this service call is terminated normally, the acquired context type (TRUE: non-task context, FALSE: task context) is returned.

Non-task contexts: cyclic handler, interrupt handler
Task contexts: task

## Return value

| Macro | Value | Description |
|---|---|---|
| TRUE | 1 | Normal completion (Non-task contexts). |
| FALSE | 0 | Normal completion (Task contexts). |

# sns_loc

## Outline

Reference CPU state.

## C format

```
BOOL    sns_loc ( void );
```

## Assembly format

```
CALL    !!_sns_loc
```

## Parameter(s)

None.

## Explanation

This service call acquires the system status type when this service call is issued (CPU locked state or CPU unlocked state).

When this service call is terminated normally, the acquired system state type (TRUE: CPU locked state, FALSE: CPU unlocked state) is returned.

Note The system enters the CPU locked state when loc_cpu or iloc_cpu is issued, and enters the CPU unlocked state when unl_cpu or iunl_cpu is issued.

## Return value

| Macro | Value | Description |
|---|---|---|
| TRUE | 1 | Normal completion (CPU locked state). |
| FALSE | 0 | Normal completion (CPU unlocked state). |

# sns_dsp

## Outline

Reference dispatching state.

## C format

```
BOOL       sns_dsp ( void );
```

## Assembly format

```
CALL    !!_sns_dsp
```

## Parameter(s)

None.

## Explanation

This service call acquires the system status type when this service call is issued (dispatching disabled state or dispatching enabled state).

When this service call is terminated normally, the acquired system state type (TRUE: dispatching disabled state, FALSE: dispatching enabled state) is returned.

Note The system enters the dispatching disabled state when dis_dsp is issued, and enters the dispatching enabled state when ena_dsp is issued.

## Return value

| Macro | Value | Description |
|---|---|---|
| TRUE | 1 | Normal completion (dispatching disabled state). |
| FALSE | 0 | Normal completion (dispatching enabled state). |

# sns_dpn

## Outline

Reference dispatch pending state.

## C format

```
BOOL    sns_dpn ( void );
```

## Assembly format

```
CALL    !!_sns_dpn
```

## Parameter(s)

None.

## Explanation

This service call acquires the system status type when this service call is issued (whether in dispatch pending state or not).

When this service call is terminated normally, the acquired system state type (TRUE: dispatch pending state, FALSE: dispatch not-pending state) is returned.

Note The dispatch pending state designates the state in which explicit execution of dispatch processing (task scheduling processing) is prohibited by issuing either the dis_dsp, loc_cpu, or iloc_cpu service call, as well as the state during which processing of a non-task is being executed.

## Return value

| Macro | Value | Description |
|---|---|---|
| TRUE | 1 | Normal completion (dispatch pending state). |
| FALSE | 0 | Normal completion (other state). |

## 12.15 System Configuration Management Functions

The following lists the service calls provided by the RI78V4 as the system configuration management functions.

Table 12-17 System Configuration Management Functions

| Service Call | Function | Origin of Service Call |
| --- | --- | --- |
| ref_ver | Reference version information. | Task, Non-task |

# CHAPTER 13 SYSTEM CONFIGURATION FILE

This chapter explains the coding method of the system configuration file required to output information files (system information table file, system information header file and interrupt information definition file) that contain data to be provided for the RI78V4.

## 13.1 Notation Method

The following shows the notation method of system configuration files.

- Character code
  Create the system configuration file using ASCII code.
  The CF78V4 distinguishes lower cases "a to z" and upper cases "A to Z".

  Note For japanese language coding, Shit-JIS codes can be used only for comments.

- Comment
  In a system configuration file, parts between /* and */ and parts from two successive slashes (//) to the line end are regarded as comments.

- Numeric
  In a system configuration file, words starting with a numeric value (0 to 9) are regarded as numeric values.
  The CF78V4 distinguishes numeric values as follows.

  Octal: Words starting with 0
  Decimal: Words starting with a value other than 0
  Hexadecimal: Words starting with 0x or 0X

  Note Elements of a word are limited to numeric values 0 to 9.

- Object name
  In a system configuration file, words starting with a letter of "a to z, A to Z", or underscore "_", within 24 characters, are regarded as object names.

  Note Elements of a word are limited to alphanumeric characters "a to z, A to Z, 0 to 9", and underscore "_".

- Symbol name
  In a system configuration file, words starting with a letter of "a to z, A to Z", or underscore "_", within 30 characters, are regarded as symbol names.

  Note 1 Elements of a word are limited to alphanumeric characters "a to z, A to Z, 0 to 9", and underscore "_".

  Note 2 The CF78V4 distinguishes the object name and symbol name according to the context in the system configuration file.

- Keywords
  The words shown below are reserved by the CF78V4 as keywords.
  Using these words for any other purpose specified is therefore prohibited.

  CLK_INTNO, CRE_CYC, CRE_DTQ, CRE_FLG, CRE_MBX, CRE_MPF, CRE_SEM, CRE_TSK, DEF_INH, .kernel_work0, .kernel_work1, .kernel_work2, .kernel_work3, MAX_PRI, null, NULL, SYS_STK, TA_ACT, TA_ASM, TA_CLR, TA_DISINT, TA_ENAINT, TA_FAR, TA_HLNG, TA_MFIFO, TA_MPRI, TA_NEAR, TA_PHS, TA_RSTR, TA_STA, TA_TFIFO, TA_TPRI, TA_WMUL, TA_WSGL

Note The CF78V4 does not call C preprocessors. Coding of preprocessing directives (#include, #define, #if, or the like) in the system configuration file is therefore prohibited.

## 13.2 Configuration Information

The configuration information that is described in a system configuration file is divided into the following two main types.

- System Information
  This information consists of fundamental data required for the RI78V4 operation.
  - System stack information
  - Task priority information
  - Clock timer interrupt source

- Static API Information
  This information consists of data for management objects required to implement the functions probided by the RI78V4.
  - Task information
  - Semaphore information
  - Eventflag information
  - Data queue information
  - Mailbox information
  - Fixed-sized memory pool information
  - Cyclic handler information
  - Interrupt handler information

### 13.2.1 Cautions

The following describes a system configuration file description format.

Figure 13-1 System Configuration File Description Format

```
-- System Information (System stack information, etc.) descriptin
/* ............ */

-- Static API Information(Task information, etc.) description
/* ............ */
```

Note Up to 40,000 lines and up to 1,000 characters per line can be written in a system configuration file.

# ref_ver

## Outline

Reference version information.

## C format

```
ER      ref_ver ( T_RVER *pk_rver );
```

## Assembly format

```
MOVW    AX, #LOWW(_pk_rver)
CALL    !!_ref_ver
```

## Parameter(s)

| I/O | Parameter | Description |
|---|---|---|
| O | T_RVER *pk_rver; | Pointer to the packet returning the version information. |

## Explanation

The service call stores version information packet (such as kernel maker's code) to the area specified by parameter *pk_rver*.

Note For details about the version information packet, refer to "12.5.9 Version information packet".

## Return value

| Macro | Value | Description |
|---|---|---|
| E_OK | 0 | Normal completion. |

## 13.3 System Information

The following describes the format that must be observed when describing the system information in the system configuration file.

The GOTHIC-FONT characters in following descriptions are the reserved words, and italic face characters are the portion that the user must write the relevant numeric value.

Items enclosed by square brackets "[ ]" can be omitted.

### 13.3.1 System stack information

Define the following item as system stack information:

1 ) System stack size: sys_stksz

Only one information item can be defined as stack information.

The following shows the system stack information format.

```
SYS_STK ( sys_stksz );
```

The items constituting the system stack information are as follows.

1 ) System stack size: *sys_stksz*

Specifies the system stack size (in bytes).

A value between 0 and 65534, aligned to a 2-byte boundary, can be specified for *sys_stksz*.

Note 1 The system stack is allocated to the .kernel_stack section.

Note 2 For details about the estimation of the system stack size, refer to See “13.5.1 System stack size“.

### 13.3.2 Task priority information

Define the following items as task priority information:

1 ) Priority range: maxtpri

The number of task priority information items that can be specified is defined as being within the range of 0 to 1.
The following shows the task priority information format.

```
[MAX_PRI ( maxtpri );]
```

The items constituting the task priority information are as follows.

1 ) Priority range: *maxtpri*

Specifies the priority range of a task (maximum value of Initial priority: itskpri, or maximum value of priority specified when issuing chg_pri).
A value between 1 and 15 can be specified for *maxtpri*.

Note If definition of this information is omitted, the task priority range is set to "15".

### 13.3.3 Clock timer interrupt source

Define the following items as clock timer interrupt source information:

1 ) Clock timer interrupt source: tim_intno

Only one information item can be defined as clock timer interrupt source information
The following shows the clock timer interrupt source information format.

```
CLK_INTNO ( tim_intno );
```

The items constituting clock timer interrupt source information are as follows.

1 ) Clock timer interrupt source: *tim_intno*

Specifies the interrupt source for a clock timer.
Only interrupt source names prescribed in the device file or only values from 0x0 to 0x7c can be specified.
If an interrupt source name is specified for *tim_intno*, the CF78V4 activation option -cpu Δ <*name*> must be specified.

## 13.4 Static API Information

The following describes the format that must be observed when describing the static API information in the system configuration file.

The GOTHIC-FONT characters in following descriptions are the reserved words, and italic face characters are the portion that the user must write the relevant numeric value, symbol name, or keyword.

Items enclosed by square brackets "[ ]" can be omitted.

### 13.4.1 Task information

Define the following items as task information:

1 ) Task name: tskid
2 ) Attribute (coding language, initial activation status, initial interrupt status): tskatr
3 ) Extended information: exinf
4 ) Start address: task
5 ) Initial priority: itskpri
6 ) Stack size: stksz
7 ) System-reserved area: stk

The number of task information items that can be specified is defined as being within the range of 1 to 127.
The following shows the task information format.

```
CRE_TSK ( tskid, { tskatr, exinf, task, itskpri, stksz, stk } );
```

The items constituting the task information are as follows.

1 ) Task name: *tskid*

Specifies the task name.
An object name can be specified for *tskid*.

Note The CF78V4 outputs to the system information header file the correspondence between the task names and IDs, in the following format. Consequently, task names can be used in the place of IDs by including the relevant system information header file using the processing program.

[ Output format to system information header file (for C) ]

```
#define tskid   ID
```

[ Output format to system information header file (for assembly language) ]

```
tskid   .EQU    ID
```

2 ) Attribute (coding language, initial activation status, initial interrupt status): *tskatr*

Specifies the attributes (coding language, initial activation status, initial interrupt status) of the task.
The keywords that can be specified for *tskatr* are TA_HLNG, TA_ASM, TA_ACT, TA_ENAINT and TA_DISINT.

[ Coding language ]
TA_HLNG: Start a processing unit through a C language interface.
TA_ASM: Start a processing unit through an assembly language interface.

[ Initial activation status ]
TA_ACT: Task is activated after the creation.

[ Initial interrupt status ]
TA_ENAINT: Enables acknowledgment of maskable interrupts.
TA_DISINT: Disables acknowledgment of maskable interrupts.

Note 1 If specification of TA_ACT is omitted, the initial task activation status is set to the "DORMANT state".

Note 2 If specification of TA_ENAINT and TA_DISINT is omitted, the initial task interrupt status is set to "interrupts acknowledgment enabled".

3 ) Extended information: *exinf*

Specifies the extended information of the task.
Values that can be specified for *exinf* are from 0 to 1048575, or symbol names written in C.

Note *exinf* is passed as an extended information to the target task when the task is activated by act_tsk or iact_tsk. The target task can therefore handle *exinf* in the same manner as handling function parameters.

4 ) Start address: *task*

Specifies the start address of the task.
Values that can be specified for *task* are symbol names written in C.

Note 1 When a task is in written in C as shown below, the value specified by this item is "func_task".

```
#include    <kernel.h>
#include    <kernel_id.h>

void
func_task ( VP_INT exinf )
{
    /* ............ */

    ext_tsk ( );
}
```

Note 2 When a task is in written in assembly language as shown below, the value specified by this item is "func_task".

```
$INCLUDE   (kernel.inc)
$INCLUDE   (kernel_id.inc)

    .PUBLIC  _func_task
    .SECTION .text, TEXT
_func_task:
    PUSH    BC
    PUSH    AX

    ; ............

    BR      !!_ext_tsk
```

5 ) Initial priority: *itskpri*

Specifies the initial priority of the task.
Values that can be specified for *itskpri* are limited to "1 to Priority range: maxtpri".

6 ) Stack size: *stksz*

Specifies the stack size (in bytes) of the task.
A value between 0 and 65534, aligned to a 2-byte boundary, can be specified for *stksz*.

Note 1 The task stack is allocated to the .kernel_stack section.

Note 2 For details about the estimation of the stack size of the task, refer to See "13.5.2 Stack size of the task".

7 ) System-reserved area: *stk*

System-reserved area.
Values that can be specified for *stk* are limited to NULL characters.

### 13.4.2 Semaphore information

Define the following items as semaphore information:

1 ) Semaphore name: semid
2 ) Attribute (queuing method): sematr
3 ) Initial resource count: isemcnt
4 ) System-reserved area: maxsem

The number of semaphore information items that can be specified is defined as being within the range of 0 to 127.
The following shows the semaphore information format.

```
CRE_SEM ( semid, { sematr, isemcnt, maxsem } );
```

The items constituting the semaphore information are as follows.

1 ) Semaphore name: *semid*

Specifies the semaphore name.
An object name can be specified for *semid*.

Note The CF78V4 outputs to the system information header file the correspondence between the semaphore names and IDs, in the following format. Consequently, semaphore names can be used in the place of IDs by including the relevant system information header file using the processing program.

[ Output format to system information header file (for C) ]

```
#define semid   ID
```

[ Output format to system information header file (for assembly language) ]

```
semid   .EQU    ID
```

2 ) Attribute (queuing method): *sematr*

Specifies the attribute (queuing method) of the semaphore.
The keywords that can be specified for *sematr* are TA_TFIFO.

[ Queuing method ]
TA_TFIFO: If a resource could not be acquired (semaphore counter is set to 0x0) when wai_sem or twai_sem is issued, the task is queued to the semaphore wait queue in the order of resource acquisition request.

3 ) Initial resource count: *isemcnt*

Specifies the initial resource count of the semaphore.
A value between 0 and 127 can be specified for *isemcnt*.

4 ) System-reserved area: *maxsem*

System-reserved area.
Values that can be specified for *maxsem* are limited to 127.

### 13.4.3 Eventflag information

Define the following items as eventflag information:

1 ) Eventflag name: flgid
2 ) Attribute (queuing method, queuing count, bit pattern clear): flgatr
3 ) System-reserved area: iflgptn

The number of eventflag information items that can be specified is defined as being within the range of 0 to 127.
The following shows the eventflag information format.

```
CRE_FLG ( flgid, { flgatr, iflgptn } );
```

The items constituting the eventflag information are as follows.

1 ) Eventflag name: *flgid*

Specifies the eventflag name.
An object name can be specified for *flgid*.

Note The CF78V4 outputs to the system information header file the correspondence between the eventflag names and IDs, in the following format. Consequently, eventflag names can be used in the place of IDs by including the relevant system information header file using the processing program.

[ Output format to system information header file (for C) ]

```
#define flgid   ID
```

[ Output format to system information header file (for assembly language) ]

```
flgid   .EQU    ID
```

2 ) Attribute (queuing method, queuing count, bit pattern clear): *flgatr*

Specifies the attributes (queuing method, queuing count, clear) of the eventflag.
The keywords that can be specified for *flgatr* are TA_TFIFO, TA_WSGL and TA_CLR.

[ Queuing method ]
TA_TFIFO: If the bit pattern of the eventflag does not satisfy the required condition when wai_flg or twai_flg is issued, the task is queued to the eventflag wait queue.

[ Queuing count ]
TA_WSGL: Only one task is allowed to be in the waiting state for the eventflag.

[ Bit pattern clear ]
TA_CLR: Bit pattern is cleared when a task is released from the waiting state for that eventflag.

Note If specification of TA_CLR is omitted, "not clear bit patterns if the required condition is satisfied" is set.

3 ) System-reserved area: *iflgptn*

System-reserved area.
Values that can be specified for *iflgptn* are limited to 0.

### 13.4.4 Data queue information

Define the following items as data qutue information:

1 ) ID number: dtqid
2 ) Attribute: dtqatr
3 ) Data count: dtqcnt, memory area name: sec_nam
4 ) Reserved for future use: dtq

The number of data queue information items that can be specified is defined as being within the range of 0 to 127.
The following shows the data queue information format.

```
CRE_DTQ (dtqid, { dtqatr, dtqcnt[:sec_nam], dtq });
```

The items constituting the data queue information are as follows.

1 ) ID number: *dtqid*

Specifies the ID number for a data queue.
A value from 0x1 to 0xff, or a name, can be specified for *dtqid*.

Note When a name is specified, the CF78V4 automatically assigns an ID number.
The CF78V4 outputs the relationship between a name and an ID number to the system information header file in the following format:

[ Output format to system information header file (for C) ]

```
#define dtqid   ID
```

[ Output format to system information header file (for assembly language) ]

```
dtqid   .EQU     ID
```

2 ) Attribute: *dtqatr*

Specifies the task queuing method for a data queue.
The keyword that can be specified for *dtqatr* is TA_TFIFO only.

TA_TFIFO: Task wait queue is in FIFO order.

3 ) Data count: *dtqcnt*, memory area name: *sec_nam*

Specifies the maximum number of data units that can be queued to the data queue area of a data queue, and the name of the memory area secured for the data queue area.
Only values from 0x0 to 0xff can be specified for *dtqcnt*, and only memory area name "kernel_work0, kernel_work1, kernel_work2, kernel_work3" can be specified for *sec_nam*.

[ section for data allocation ]
kernel_work0 : allocates data queue to ".kernel_work0" section
kernel_work1 : allocates data queue to ".kernel_work1" section
kernel_work2 : allocates data queue to ".kernel_work2" section
kernel_work3 : allocates data queue to ".kernel_work3" section

4 ) Reserved for future use: *dtq*

System-reserved area.
Values that can be specified for *dtq* are limited to NULL characters.

### 13.4.5 Mailbox information

Define the following items as mailbox information:

1 ) Mailbox name: mbxid
2 ) Attribute (queuing method): mbxatr
3 ) System-reserved area: maxmpri
4 ) System-reserved area: mprihd

The number of mailbox information items that can be specified is defined as being within the range of 0 to 127.
The following shows the mailbox information format.

```
CRE_MBX ( mbxid, { mbxatr, maxmpri, mprihd } );
```

The items constituting the mailbox information are as follows.

1 ) Mailbox name: *mbxid*

Specifies the mailbox name.
An object name can be specified for *mbxid*.

Note The CF78V4 outputs to the system information header file the correspondence between the mailbox names and IDs, in the following format. Consequently, mailbox names can be used in the place of IDs by including the relevant system information header file using the processing program.

[ Output format to system information header file (for C) ]

```
#define mbxid   ID
```

[ Output format to system information header file (for assembly language) ]

```
mbxid   .EQU    ID
```

2 ) Attribute (queuing method): *mbxatr*

Specifies the attributes (task queuing method, message queuing method) of the mailbox.
The keywords that can be specified for *mbxatr* are TA_TFIFO, TA_MFIFO and TA_MPRI.

[Task queuing method ]
TA_TFIFO: If the message could not be received from the mailbox (no messages were queued in the wait queue) when rcv_mbx or trcv_mbx is issued, the task is queued to the mailbox wait queue in the order of message reception request.

[ Message queuing method ]
TA_MFIFO: If a task is not queued to the mailbox wait queue when snd_mbx is issued, the message is queued to the mailbox wait queue in the order of message transmission request.
TA_MPRI: If a task is not queued to the mailbox wait queue when snd_mbx is issued, the message is queued to the mailbox wait queue in the order of message priority.

3 ) System-reserved area: *maxmpri*

System-reserved area.
Values that can be specified for *maxmpri* are limited to 0.

4 ) System-reserved area: *mprihd*

System-reserved area.
The keywords that can be specified for *mprihd* are NULL.

### 13.4.6 Fixed-sized memory pool information

Define the following items as fixed-sized memory pool information:

1 ) Fixed-sized memory pool name: mpfid
2 ) Attribute (queuing method): mpfatr
3 ) Total number of memory blocks: blkcnt
4 ) Memory block size: blksz
5 ) Section name: sec_nam
6 ) System-reserved area: mpf

The number of fixed-sized memory pool information items that can be specified is defined as being within the range of 0 to 127.

The following shows the fixed-sized memory pool information format.

```
CRE_MPF ( mpfid, { mpfatr, blkcnt, blksz[:sec_nam], mpf } );
```

The items constituting the fixed-sized memory pool information are as follows.

1 ) Fixed-sized memory pool name: *mpfid*

Specifies the fixed-sized memory pool name.
An object name can be specified for *mpfid*.

Note The CF78V4 outputs to the system information header file the correspondence between the fixed-sized memory pool names and IDs, in the following format. Consequently, fixed-sized memory pool names can be used in the place of IDs by including the relevant system information header file using the processing program.

[ Output format to system information header file (for C) ]

```
#define mpfid   ID
```

[ Output format to system information header file (for assembly language) ]

```
mpfid   .EQU    ID
```

2 ) Attribute (queuing method): *mpfatr*

Specifies the attribute (queuing method) of the fixed-sized memory pool.
The keywords that can be specified for *mpfatr* are TA_TFIFO.

[ Queuing method ]
TA_TFIFO: If a memory block could not be acquired (no available memory blocks exist) when get_mpf or tget_mpf is issued, the task is queued to the fixed-sized memory pool wait queue in the order of memory block acquisition request.

3 ) Total number of memory blocks: *blkcnt*

Specifies the total number of memory blocks.
A value between 1 and 16383 can be specified for *blkcnt*.

4 ) Memory block size: *blksz*

Specifies the memory block size (in bytes).
A value between 4 and 65534, aligned to a 2-byte boundary, can be specified for *blksz*.

5 ) Section name: *sec_nam*

Specifies where the fixed-sized memory pool is to be allocated.
Values that can be specified for *sec_nam* are limited to kernel_work0, kernel_work1, kernel_work2, or kernel_work3.

[ Fixed-sized memory pool allocation section ]
kernel_work0: Allocates the fixed-sized memory pool to the .kernel_work0 section.
kernel_work1: Allocates the fixed-sized memory pool to the .kernel_work1 section.
kernel_work2: Allocates the fixed-sized memory pool to the .kernel_work2 section.
kernel_work3: Allocates the fixed-sized memory pool to the .kernel_work3 section.

Note If specification of *seg_nam* is omitted, the fixed-sized memory pool is allocated to the .kernel_work0 section.

6 ) System-reserved area: *mpf*

System-reserved area.
Values that can be specified for *mpf* are limited to NULL characters.

### 13.4.7 Cyclic handler information

Define the following items as cyclic handler information:

1 ) Cyclic handler name: cycid
2 ) Attribute (coding language, initial activation status, saving activation phase): cycatr
3 ) System-reserved area: exinf
4 ) Start address: cychdr
5 ) Activation cycle: cyctim
6 ) Activation phase: cycphs

The number of cyclic handler information items that can be specified is defined as being within the range of 0 to 127.
The following shows the cyclic handler information format.

```
CRE_CYC ( cycid, { cycatr, exinf, cychdr, cyctim, cycphs } );
```

The items constituting the cyclic handler information are as follows.

1 ) Cyclic handler name: *cycid*

Specifies the cyclic handler name.
An object name can be specified for *cycid*.

Note The CF78V4 outputs to the system information header file the correspondence between the cyclic handler names and IDs, in the following format. Consequently, cyclic handler names can be used in the place of IDs by including the relevant system information header file using the processing program.

[ Output format to system information header file (for C) ]

```
#define cycid   ID
```

[ Output format to system information header file (for assembly language) ]

```
cycid   .EQU    ID
```

2 ) Attribute (coding language, initial activation status, saving activation phase): *cycatr*

Specifies the attributes (coding language, initial activation status) of the cyclic handler.
The keywords that can be specified for *cycatr* are TA_HLNG, TA_ASM and TA_STA, TA_PHS.

[ Coding language ]
TA_HLNG: Start a processing unit through a C language interface.
TA_ASM: Start a processing unit through an assembly language interface.

[ Initial operation status ]
TA_STA: Cyclic handler is in an operational state after the creation.

[ Saving activation phase ]
TA_PHS: Saves activation phase.

Note If specification of TA_STA is omitted, the cyclic handler initial activation status is set to "non-operational state (STP state)".

3 ) System-reserved area: *exinf*

System-reserved area.
Values that can be specified for *exinf* are limited to 0.

4 ) Start address: *cychdr*

Specifies the start address of the cyclic handler.
Values that can be specified for *cychdr* are symbol names written in C.

Note 1 When the cyclic handler is in written in C as shown below, the value specified by this item is "func_cychdr".

```
#include    <kernel.h>
#include    <kernel_id.h>

void
func_cychdr ( void )
{
    /* ............ */

    return;
}
```

Note 2 When the cyclic handler is in written in assembly language as shown below, the value specified by this item is "func_cychdr".

```
$INCLUDE    (kernel.inc)
$INCLUDE    (kernel_id.inc)

    .PUBLIC  _func_cychdr
    .SECTION .text, TEXT
_func_cychdr:
    ; ............

    RET
```

5 ) Activation cycle: *cyctim*

Specifies the activation cycle (unit: ticks) of the cyclic handler.
A value between 1 and 4294967295 can be specified for *cyctim*.

6 ) Activation phase: *cycphs*

Specifies the activation phase (in millisecond) for a cyclic handler.
A value from 0x1 to 0x7fffffff (aligned to ‘clkcyc’ multiple values) can be specified for *cycphs*.

Note 1 In the RI78V4, the initial activation phase means the relative interval from when generation of s cyclic handler is completed until the first activation request is issued.

Note 2 If a value other than an integral multiple of the base clock cycle defined in Clock timer interrupt source is specified for *cycphs*, the CF78V4 assumes that an integral multiple is specified and performs processing.

### 13.4.8 Interrupt handler information

Define the following items as interrupt handler information:

1 ) Interrupt source: inhno
2 ) Attribute: inhatr
3 ) Start address: inthdr

The number of items that can be defined as interrupt handler information is limited to one for each interrupt source.
The following shows the interrupt handler information format.

```
DEF_INH (inhno, { inhatr, inthdr });
```

The items constituting the interrupt handler information are as follows.

1 ) Interrupt source: *inhno*

Specifies the interrupt source for an interrupt handler.
Only interrupt source names prescribed in the device file or only values from 0x0 to 0x7c can be specified.
If an interrupt source name is specified for *inhno*, the CF78V4 activation option -cpu Δ <*name*> must be specified.

2 ) Attribute: *inhatr*

Specifies the language used to describe an interrupt handler.
The keyword that can be specified for *inhatr* is TA_HLNG or TA_ASM.

[ Coding language ]

TA_HLNG: Start an interrupt handler through a C language interface.
TA_ASM: Start an interrupt handler through an assembly language interface.

[ Allocation ]

TA_NEAR : NEAR allocation
TA_FAR : FAR allocation

3 ) Start address: *inthdr*

Specifies the start address of the interrupt handler.
Values that can be specified for *inthdr* are symbol names written in C.

Note 1 When a interrupt handler is in written in C as shown below, the value specified by this item is "func_inthdr".

```
#include        <kernel.h>
#include        <kernel_id.h>

void
func_inthdr ( void )
{
        ............
        ............

        return;
}
```

Note 2 When a interrupt handler is in written in assembly language as shown below, the value specified by this item is "func_inthdr".

```
$INCLUDE       (kernel.inc)
$INCLUDE       (kernel_id.inc)

        .PUBLIC  _func_inthdr
        .SECTION .text, TEXT
_func_inthdr:
        ............
        ............

        RET
```

## 13.5 Stack Size Estimation

### 13.5.1 System stack size

The formula for calculating the system stack size is shown below.

[Expression 1: System stack size]
sys_stk = MAX(sys_stkA, sys_stkB, sys_stkC) + 2 (bytes)

[Expression 2: System stack size use pattern A]
sys_stkA = tsksvc + int0 + int1 + int2 + int3

[Expression 3: System stack size use pattern B]
sys_stkB = Size used by user in idle routine

[Expression 4: System stack size use pattern C]
sys_stkC = Size used by user in initialization routine

[Expression 5: Maximum size of system stack used during service call executed by task]
Maximum size of system stack used during service call executed by task

[Expression 6: Size of int0, int1]
Intx = Maximum size of interrupts used by stack in interrupts of level x
= Size used by user in interrupts

[Expression 7: Size of int2, int2]
intx = Maximum size of interrupts used by stack in interrupts of level x
= Size used by user in interrupts + allsvc + 18

[Expression 8: Total size used by system calls used in interrupt]
allsvc = For service call arguments + For internal processing by program issued the service call + For system stack internal processing

Specify the system stack size in the system configuration file. Note, however, that the size that is actually secured is the value specified in the configurator + 2 bytes. Consequently, the value that is actually specified in the system configuration file is the sys_stk value calculated in expression 1 minus 2 bytes.

We recommend specifying a system stack size higher than the estimate in order to reduce the danger of a stack overflow.

The example is shown below.

[Conditions]

- Execute a pol_flg service call from task "task1".
- Execute a snd_mbx service call from task "task2".
- Interrupt int0 is a level-0 interrupt process not managed by the OS. The stack is not used in the interrupt.
- Interrupt int2 is a level-2 OS interrupt handler. Execute the snd_mbx service call, and use 12 bytes of stack in the interrupt.
- Interrupt int3A is a level-3 OS interrupt handler. Execute the pol_flg service call, and use 16 bytes of stack in the interrupt.
- Interrupt int3B is a level-3 OS interrupt handler. Execute Timer_Handler, the stack is not used in the interrupt.
- Idle "idl" does not use the stack.
- The initialization routine "ini" uses 24 bytes of stack in the routine.

[Expression]

tsksvc = MAX(size of system stack used by pol_flg, size of system stack used by snd_mbx)
= MAX(16,8) = 16 bytes

int0 = 0 + 0 = 0 byte

int1 = undefined = 0 byte

int2 = 12 + ( 0 + 6 + 4 ) + 18 = 40 bytes

int3 = MAX(int3A, int3B) = MAX(56,32) = 56 bytes

int3A = 16 + ( 0 + 6 + 16 ) + 18 = 56 bytes

int3B = 0 + ( 0 + 0 + 14 ) + 18 = 32 bytes

sys_stkA = tsksvc + int0 + int1 + int2 + int3
= 16 ＋ 0 ＋ 0 ＋ 40 ＋ 56
= 112 bytes

Note This is the max in sys_stkA/B/C, so after this size or greater is secured.

sys_stkB = Stack size used by user in idle routine = 0 byte

sys_stkC = Stack size used by user in initialization routine = 20 bytes

sys_stk = MAX(sys_stkA, sys_stkB, sys_stkC) ＋ 2
= MAX(112, 0, 20)
= 112 + 2 = 114 bytes

The system stack size will be the 112 bytes of sys_stkA.
The size specified in the system configuration file will be 112 bytes.

Note Below is shown the stack size used in service calls/functions used in the example.

| | For Service Call Arguments | For Internal Processing by Program Issued the Service Call | For System Stack Internal Processing |
|---|---|---|---|
| pol_flg | 0 | 6 | 16 |
| twai_flg | 4 | 6 | 16 |
| snd_mbx | 0 | 6 | 4 |
| Timer_Handler function | 0 | — | 14 |

### 13.5.2 Stack size of the task

The formula for calculating the stack size of the task is shown below.

[Expression 1: No interrupts generated in task]
Task stack size = size used by user + service-call argument size + 6 (bytes)

[Expression 2: Interrupts generated in task]
Task stack size = size used by user + service-call argument size + 6 + 20 (bytes)

Specify the task stack size in the system configuration file. Note, however, that the size that is actually secured is the value specified in the configurator 6 bytes. Consequently, the value that is actually specified in the system configuration file is the sys_stk value calculated in expression 1 or expression 2 minus 6 bytes.

These 6 bytes include the stack size used when system calls are issued. Note, however, that the stack size used when issuing system calls must secure the size used by the user in addition to the 6 bytes of argument stack size. The argument stack sized used by each service call is different. Table 12-1 summarizes these sizes.

The task stack size is the largest stack size used in the task in question. For this reason, if there is a service call with an argument stack of 4 bytes, and another with 8 bytes, then the pattern that uses the most stack - 8 bytes - will be secured.

The above material refers to tasks where interrupts are not accepted (all interrupts are disabled). An additional 18 bytes must be secured for tasks where interrupts are accepted.

Note that these 20 bytes include the stack size when the _kernel_int_entry function is called (required to be called when an interrupt starts). _kernel_int_entry only retires the 20 bytes of data from the stack, it does not replace it. The data is recovered upon the call to the _kernel_int_exit function, which must be called when the interrupt ends.

Example 1 Task "task1" uses the twai_flg and snd_mbx service calls, and has no other functions or processes that use the stack.

If interrupts are not accepted in the task, interrupts are not accepted in task1, so Expression 1 is the formula for calculating stack usage.
Because there are no functions or processes that use the stack, the size used by the user is 0.
When the size of arguments to all service calls is investigated, the results are as shown below.

Service-call argument size (twai_flg) = 4 bytes
Service-call argument size (snd_mbx) = 0 bytes

The largest stack size is used in the call to twai_flg, so this is specified in Expression 1.

Task stack size = size used by user + service-call argument size (twai_flg) + 6
= 0 + 4 + 6
= 10 bytes

The size specified in the system configuration file will be the above minus 6 bytes, which equals 4 bytes.

Example 2 In task "task1", function A (using 12 bytes of stack) makes a twai_flg service call, and function B (using 20 bytes of stack) makes a snd_mbx service call.

Since interrupts are accepted in the task, Expression 2 is used as the calculation formula. List the patterns in order to find the one that uses the most stack.

Pattern A = size used by user (for function A) + service-call argument size (twai_flg) + 6 + 20
= 12 + 4 + 6 + 20
= 42 bytes

Pattern B = size used by user (for function B) + service-call argument size (snd_mbx) + 6 + 20
= 20 + 0 + 6 + 20
= 26 bytes

Compare pattern B with pattern A. The pattern that uses the most stack is pattern A, at 42 bytes.
The size specified in the system configuration file will be the above minus 6 bytes, which equals 36 bytes.

## 13.6 Description Examples

The following describes an example for coding the system configuration file.

Figure 13-2 Example of System Configuration File

```
-- System Information description
SYS_STK ( 256 );
MAX_PRI ( 15 );

-- Static API Information description
CRE_TSK ( ID_tsk, { TA_HLNG | TA_ACT | TA_DISINT, 0xa, func_task, 1 256, NULL } );
CRE_TSK ( ID_tskA, { TA_HLNG | TA_ACT, 0x14, func_taskA, 2, 256, NULL } );
CRE_TSK ( ID_tskB, { TA_ASM | TA_ENAINT, 0x1e, func_taskB, 3, 512, NULL } );

CRE_SEM ( ID_semA, { TA_TFIFO, 0, 127 } );
CRE_SEM ( ID_semB, { TA_TFIFO, 127, 127 } );

CRE_FLG ( ID_flgA, { TA_TFIFO | TA_WSGL | TA_CLR, 0 } );
CRE_FLG ( ID_flgB, { TA_TFIFO | TA_WSGL, 0 } );

CRE_DTQ ( ID_DTQ1, { TA_TFIFO,20:kernel_work1, NULL } );

CRE_MBX ( ID_mbxA, { TA_TFIFO | TA_MFIFO, 0, NULL } );
CRE_MBX ( ID_mbxB, { TA_TFIFO | TA_MPRI, 0, MULL } );

CRE_MPF ( ID_mpfA, { TA_TFIFO, 10, 8:kernel_work1, NULL } );
CRE_MPF ( ID_mpfB, { TA_TFIFO, 8, 16, NULL } );

CRE_CYC ( ID_cycA, { TA_HLNG | TA_STA, 0, func_cychdrA, 1, 0x50 } );
CRE_CYC ( ID_cycB, { TA_ASM, 0, func_cychdrB, 2, 0x100 } );

DEF_INH ( INTP0, { TA_HLNG | TA_FAR, inthdr0 } );
DEF_INH ( INTP1, { TA_HLNG | TA_NEAR, inthdr1 } );
```

# CHAPTER 14 CONFIGURATOR CF78V4

This chapter explains configurator CF78V4, which is provided by the RI78V4 as a utility tool useful for system construction.

## 14.1 Outline

To build systems (load module) that use functions provided by the RI78V4, the information storing data to be provided for the RI78V4 is required.

Since information files are basically enumerations of data, it is possible to describe them with various editors.

Information files, however, do not excel in descriptiveness and readability; therefore substantial time and effort are required when they are described.

To solve this problem, the RI78V4 provides a utility tool (configurator CF78V4) that converts a system configuration file which excels in descriptiveness and readability into information files.

The CF78V4 reads the system configuration file as a input file, and then outputs information files.

The information files output from the CF78V4 are explained below.

- System information table file
  An information file that stores data required for the operation of the RI78V4.

- System information header file
  An information file that stores matching between ID numbers and object names (e.g. task, and semaphore names) described in the system configuration file.
  The CF78V4 can output two types of system information header files for C and assembly languages.

- Interrupt information definition file
  An information file that stores related interrupt handler described in the system configuration file.

## 14.2 Activation Method

### 14.2.1 Activating from command line

The following is how to activate the CF78V4 from the command line.

Note that, in the examples below, "C>" indicates the command prompt, "Δ" indicates pressing of the space key, and "<Enter>" indicates pressing of the enter key.

The activation options enclosed in "[ ]" can be omitted.

```
C> cf78v4.exe Δ [@command file] Δ [-cpu Δ <name>] Δ [-devpath=path]
   Δ [-i Δ <SIT file> |-ni] Δ [-dc Δ <C header file> | -ndc]
   Δ [-da Δ <ASM header file> | -nda] Δ [-V] Δ [-help] Δ <CF file> <Enter>
```

The details of each activation option are explained below:

- @command file
  Specifies the command file name to be input.

  If omitted The activation options specified on the command line is valid.

  Note 1 Specify the input file name “command file” within 255 characters including the path name.

  Note 2 For the details about the command file, refer to “14.2.3 Command file”.

- -cpu Δ *<name>*
  Specifies type specification names of target device.

  If omitted If this activation option is not specified, the CF78V4 does not load the device file. As a result, definitions using interrupt source names defined in the device file can no longer be used in the system configuration file.

- -devpath=*path*
  Retrieves the device file corresponding to the target device specified with -cpu Δ <name> from the path folder.

  If omitted The device file is retrieved for the current folder.

- -iΔ<SIT file>
  Specifies the system information table file name to be output.

  If omitted If omitted, the CF78V4 interprets it that -iΔsit.asm is specified.

  Note Specify the output file name “<SIT file>” within 255 characters including the path name.

- -ni
  Disables output of the system information table file.

  If omitted If omitted, the CF78V4 interprets it that -iΔsit.asm is specified.

- -dcΔ<C header file>
  Specifies the system information header file (for C language) name to be output.

  If omitted If omitted, the CF78V4 interprets it that -dcΔkernel_id.h is specified.

  Note Specify the output file name “<SIT file>” within 255 characters including the path name.

- -ndc
  Disables output of the system information header file (for C language).

  If omitted  If omitted, the CF78V4 interprets it that -dcΔkernel_id.h is specified.

- -daΔ<ASM header file>
  Specifies the system information header file (for assembly language) name to be output.

  If omitted  If omitted, the CF78V4 interprets it that -daΔkernel_id.inc is specified.

  Note  Specify the output file name “<ASM header file>” within 255 characters including the path name.

- -nda
  Disables output of the system information header file (for assembly language).

  If omitted  If omitted, the CF78V4 interprets it that -daΔkernel_id.inc .inc is specified.

- -V
  Outputs version information for the CF78V4 to the standard output.

  Note  If this activation option is specified, the CF78V4 handles other activation options as invalid options and suppresses outputting of information files.

- -help
  Outputs the usage of the activation options for the CF78V4 to the standard output.

  Note  If this activation option is specified, the CF78V4 handles other activation options as invalid options and suppresses outputting of information files.

- <CF file>
  Specifies the system configuration file name to be input.

  Note 1  Specify the input file name “<CF file>” within 255 characters including the path name.
  Note 2  This input file name can be omitted only when -V or -help is specified.

### 14.2.2 Activating from CS+

This is started when the CS+ performs a build, in accordance with the setting on the Property panel, on the [System Configuration File Related Information] tab.

### 14.2.3 Command file

The CF78V4 performs command file support from the objectives that eliminate specified probable activation option character count restrictions in the command lines.

Description formats of the command file are described below.

1 ) Comment lines
Lines that start with # are treated as comment lines.

2 ) Dilimiting activation options
Delimit activation options using a space code, tab code, or a linefeed code.

Note For activation options consist of the *-xxx* part and parameter part, like "-iΔ<SIT file>", "-dcΔ<C header file>", and "-daΔ<ASM header file>", delimit the *-xxx* part and parameter part using a space code, tab code, or a linefeed code.
When specifying a folder name that includes a space code in the parameter part, enclose the parameter part using double-quotation marks (") as shown in Figure 14-1.

3 ) Maximum number of characters
Up to 50 lines and up to 4,096 characters per line can be coded in a command file.

The following shows an example of activation option coding whereby "system configuration file CF_file.cfg is loaded from the current folder, system information table file sit_file.asm is output to a folder in C:\Program Files\tmp, system information header file C_header.h (for C) is output to a folder in C:\tmp, system information header file ASM_header.inc (for assembly language) is output to a folder in C:\tmp".

Figure 14-1 Example of Command File Description

```
# Command File
-i "C:\Program Files\tmp\sit_file.asm"
-dc C:\tmp\C_header.h
-da
"C:\tmp\ASM_header.inc"
CF_file.cfg
```

### 14.2.4 Command input examples

The following shows the CF78V4 command input examples.

In these examples, "C>" indicates the command prompt, "Δ" indicates the space key input, and "<Enter>" indicates the ENTER key input.

1 ) After loading command file cmd_file from the current folder, the activation option defined in cmd_file is executed.

```
C> cf78v4.exe Δ @cmd_file <Enter>
```

2 ) After loading system configuration file CF_file.cfg from the current folder, system information table file sit_file.asm, the system information header file C_header.h (for C) and system information header file ASM_header.inc (for assembly language) are output to the current folder (specified device name is R5F10A6A,and the path for device file is ”C:\Program Files\Renesas Electronics\CS+\CC\Device\RL78\Devicefile”.
).

```
C> cf78v4.exe Δ -cpu Δ R5F10A6A Δ
   -devpath=”C:\Program Files\Renesas Electronics\CS+\CC\Device\RL78\Devicefile”
   Δ -iΔsit_file.asm Δ -dc Δ C_header.h Δ -da Δ ASM_header.inc
   Δ CF_file.cfg<Enter>
```

3 ) After loading system configuration file CF_file.cfg from the current folder, system information table file sit.asm, the system information header file kernel_id.h (for C) and system information header file kernel_id.inc (for assembly language) are output to the current folder.

```
C> cf78v4.exe Δ CF_file.cfg <Enter>
```

4 ) After loading system configuration file CF_file.cfg from a folder in C:\tmp, system information table file sit_file.asm, the system information header file C_header.h (for C) is output to a folder in C:\tmp.

```
C> cf78v4.exe Δ -i Δ C:\tmp\sit_file.asm Δ -dc Δ C:\tmp\C_header.h Δ -nda Δ
   C:\tmp\CF_file.cfg <Enter>
```

5 ) After loading system configuration file CF_file.cfg from a folder in C:\tmp, the system information table file sit_file.asm is output to a folder in C:\Program Files\tmp.

```
C> cf78v4.exeΔ-i Δ ”C:\Program Files\tmp\sit_file.asm" Δ -ndc Δ -nda Δ
   C:\tmp\CF_file.cfg <Enter>
```

6 ) CF78V4 version information is output to the standard output.

```
C> cf78v4.exe Δ -V <Enter>
```

7 ) Information related to the CF78V4 activation option (type, usage, or the like) is output to the standard output.

```
C> cf78v4.exe Δ -help <Enter>
```

# APPENDIX A WINDOW REFERENCE

This appendix explains the window/panels that are used when the activation option for the CF78V4 is specified from the integrated development environment platform “CS+”.

## A.1 Description

The following shows the list of window/panels.

Table A-1 List of Window/Panels

| Window/Panel Name | Function Description |
|---|---|
| Main window | This is the first window to be open when the CS+ is launched. |
| Project Tree panel | This panel is used to display the project components in tree view. |
| Property panel | This panel is used to display the detailed information on the Realtime OS node, system configuration file, or the like that is selected on the Project Tree panel and change the settings of the information. |

# Main window

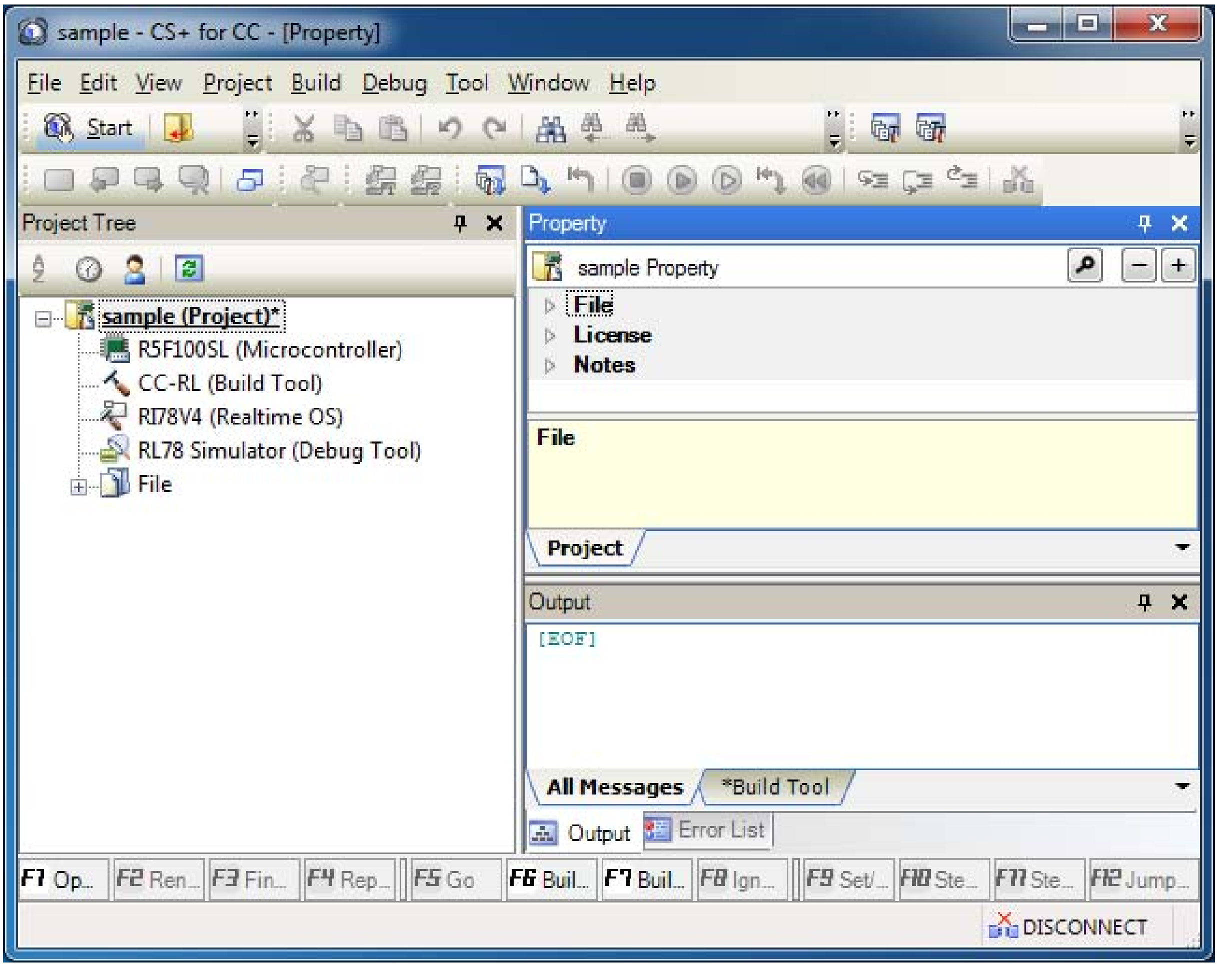
## Outline

This is the first window to be open when the CS+ is launched.
This window is used to control the user program execution and open panels for the build process.

This window can be opened as follows:

- Select Windows [start] -> [All programs] -> [Renesas Electronics CS+] -> [CS+]

## Display image



## Explanation of each area

1 ) Menu bar

Displays the menus relate to realtime OS.
Contents of each menu can be customized in the User Setting dialog box.

- [View]

| | | |
|---|---|---|
| Realtime OS | | The [View] menu shows the cascading menu to start the tools of realtime OS. |
| | Resource Information | Opens the Realtime OS Resource Information panel.<br>Note that this menu is disabled when the debug tool is not connected. |
| | Task Analyzer | Opens the Task Analyzer window.<br>Note that this menu is disabled when the debug tool is not connected. |

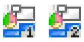
2 ) Toolbar

Displays the buttons relate to realtime OS.
Buttons on the toolbar can be customized in the User Setting dialog box. You can also create a new toolbar in the same dialog box.

- Realtime OS toolbar

| | |
|---|---|
|  | Opens the Realtime OS Resource Information panel.<br>Note that this button is disabled when the debug tool is not connected. |

- Task Analyzer

| | |
|---|---|
|  | Opens the Realtime OS Task Analyzer panel.<br>Note that this button is disabled when the debug tool is not connected. |

3 ) Panel display area

The following panels are displayed in this area.

- Project Tree panel
- Property panel
- Output panel

See the each panel section for details of the contents of the display.

Note See “CS+ Integrated Development User's Manual: RL78 Build” for details about the Output panel.

# Project Tree panel

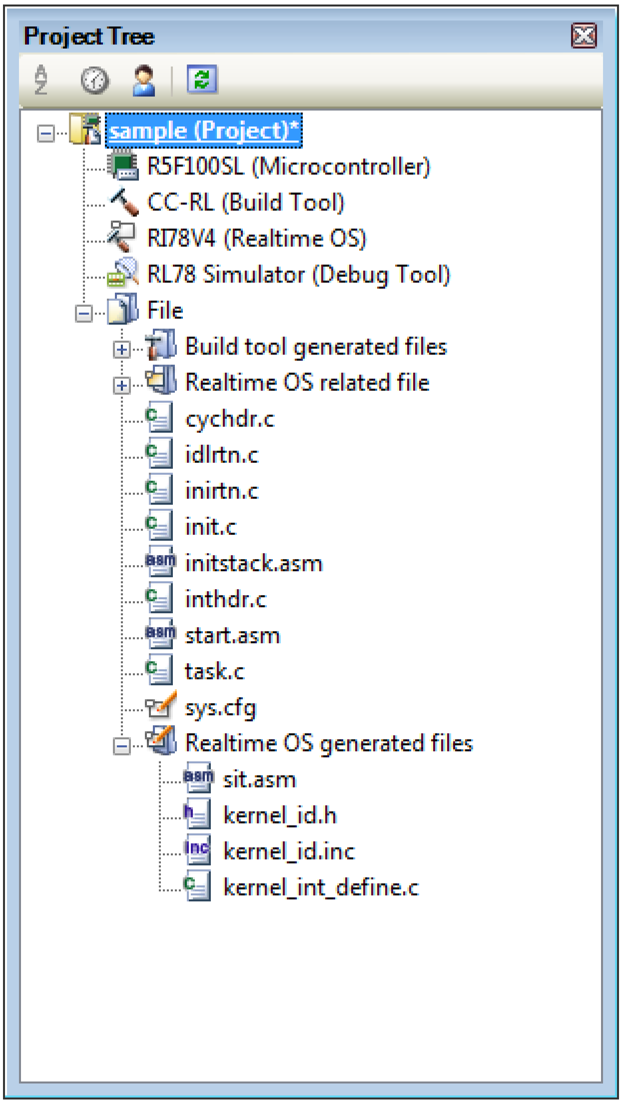
## Outline

This panel is used to display the project components such as Realtime OS node, system configuration file, etc. in tree view.

This panel can be opened as follows:

- From the [View] menu, select [Project Tree].

## Display image



## Explanation of each area

1 ) Project tree area

Project components are displayed in tree view with the following given node.

| Node | Description |
|---|---|
| RI78V4(Realtime OS) (referred to as "Realtime OS node") | Realtime OS to be used. |
| *xxx*.cfg | System configuration file. |
| Realtime OS generated files (referred to as "Realtime OS generated files node") | The following information files appear directly below the node created when a system configuration file is added.<br>- System information table file (.asm)<br>- System information header file (for C language) (.h)<br>- System information header file (for assembly language) (.inc)<br>This node and files displayed under this node cannot be deleted directly. This node and files displayed under this node will no longer appear if you remove the system configuration file from the project. |
| Realtime OS related files (referred to as "Realtime OS related files node") | The following information files appear directly below the node.<br>- Trace information file (trcinf.c)<br>This node and files displayed under this node cannot be deleted. |

## Context menu

1 ) When the Realtime OS node or Realtime OS generated files node is selected

| | |
|---|---|
| Property | Displays the selected node's property on the Property panel. |

2 ) When the system configuration file or an information file is selected

| | |
|---|---|
| Assemble | Assembles the selected assembler source file.<br>Note that this menu is only displayed when a system information table file is selected.<br>Note that this menu is disabled when the build tool is in operation. |
| Open | Opens the selected file with the application corresponds to the file extension.<br>Note that this menu is disabled when multiple files are selected. |
| Open with Internal Editor... | Opens the selected file with the Editor panel.<br>Note that this menu is disabled when multiple files are selected. |
| Open with Selected Application... | Opens the Open with Program dialog box to open the selected file with the designated application.<br>Note that this menu is disabled when multiple files are selected. |
| Open Folder with Explorer | Opens the folder that contains the selected file with Explorer. |
| Add | Shows the cascading menu to add files and category nodes to the project. |
| Add File... | Opens the Add Existing File dialog box to add the selected file to the project. |

| | |
|---|---|
| Add New File... | Opens the Add File dialog box to create a file with the selected file type and add to the project. |
| Add New Category | Adds a new category node at the same level as the selected file. You can rename the category.<br>This menu is disabled while the build tool is running, and if categories are nested 20 levels. |
| Remove from Project | Removes the selected file from the project.<br>The file itself is not deleted from the file system.<br>Note that this menu is disabled when the build tool is in operation. |
| Copy | Copies the selected file to the clipboard.<br>When the file name is in editing, the characters of the selection are copied to the clipboard. |
| Paste | This menu is always disabled. |
| Rename | You can rename the selected file.<br>The actual file is also renamed. |
| Property | Displays the selected file's property on the Property panel. |

# Property panel

## Outline

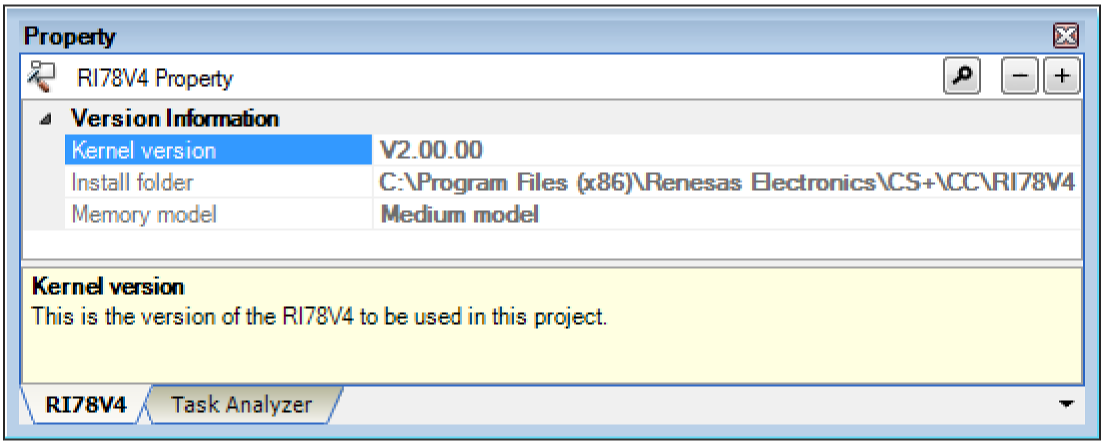
This panel is used to display the detailed information on the Realtime OS node, system configuration file, or the like that is selected on the Project Tree panel by every category and change the settings of the information.

This panel can be opened as follows:

- On the Project Tree panel, select the Realtime OS node, system configuration file, or the like, and then select the [View] menu -> [Property] or the [Property] from the context menu.

Note When either one of the Realtime OS node, system configuration file, or the like on the Project Tree panel while the Property panel is opened, the detailed information of the selected node is displayed.

## Display image




## Explanation of each area

1 ) Selected node area

Display the name of the selected node on the Project Tree panel.
When multiple nodes are selected, this area is blank.

2 ) Detailed information display/change area

In this area, the detailed information on the Realtime OS node, system configuration file, or the like that is selected on the Project Tree panel is displayed by every category in the list. And the settings of the information can be changed directly.
Mark ⊟ indicates that all the items in the category are expanded. Mark ⊞ indicates that all the items are collapsed.
You can expand/collapse the items by clicking these marks or double clicking the category name
See the section on each tab for the details of the display/setting in the category and its contents.

3 ) Property description area

Display the brief description of the categories and their contents selected in the detailed information display/change area.

4 ) Tab selection area

Categories for the display of the detailed information are changed by selecting a tab.
In this panel, the following tabs are contained (see the section on each tab for the details of the display/setting on the tab).

- When the Realtime OS node is selected on the Project Tree panel
  - [RI78V4] tab
- When the system configuration file is selected on the Project Tree panel
  - [System Configuration File Related Information] tab
  - [File Information] tab
- When the Realtime OS generated files node is selected on the Project Tree panel
  - [Category Information] tab
- When the system information table file is selected on the Project Tree panel
  - [Build Settings] tab
  - [Individual Assemble Options] tab
  - [File Information] tab
- When the system information header file is selected on the Project Tree panel
  - [File Information] tab
- When the interrupt information definition file is selected on the Project Tree panel
  - [File Information] tab
- When the trace information file (*trcinf.c*) is selected on the Project Tree panel
  - [File Information] tab

Note1 See “CS+ Integrated Development Environment User’s Manual: RL78 Build” for details about the [File Information] tab, [Category Information] tab, [Build Settings] tab, and [Individual Assemble Options] tab.

Note2 When multiple components are selected on the Project Tree panel, only the tab that is common to all the components is displayed. If the value of the property is modified, that is taken effect to the selected components all of which are common to all.

## [Edit] menu (only available for the Project Tree panel)

| | |
|---|---|
| Undo | Cancels the previous edit operation of the value of the property. |
| Cut | While editing the value of the property, cuts the selected characters and copies them to the clip board. |
| Copy | Copies the selected characters of the property to the clip board. |
| Paste | While editing the value of the property, inserts the contents of the clip board. |
| Delete | While editing the value of the property, deletes the selected character string. |
| Select All | While editing the value of the property, selects all the characters of the selected property. |

## Context menu

| | |
|---|---|
| Undo | Cancels the previous edit operation of the value of the property. |
| Cut | While editing the value of the property, cuts the selected characters and copies them to the clip board. |
| Copy | Copies the selected characters of the property to the clip board. |
| Paste | While editing the value of the property, inserts the contents of the clip board. |
| Delete | While editing the value of the property, deletes the selected character string. |
| Select All | While editing the value of the property, selects all the characters of the selected property. |
| Reset to Default | Restores the configuration of the selected item to the default configuration of the project.<br>For the [Individual Assemble Options] tab, restores to the configuration of the general option. |
| Reset All to Default | Restores all the configuration of the current tab to the default configuration of the project.<br>For the [Individual Assemble Options] tab, restores to the configuration of the general option. |

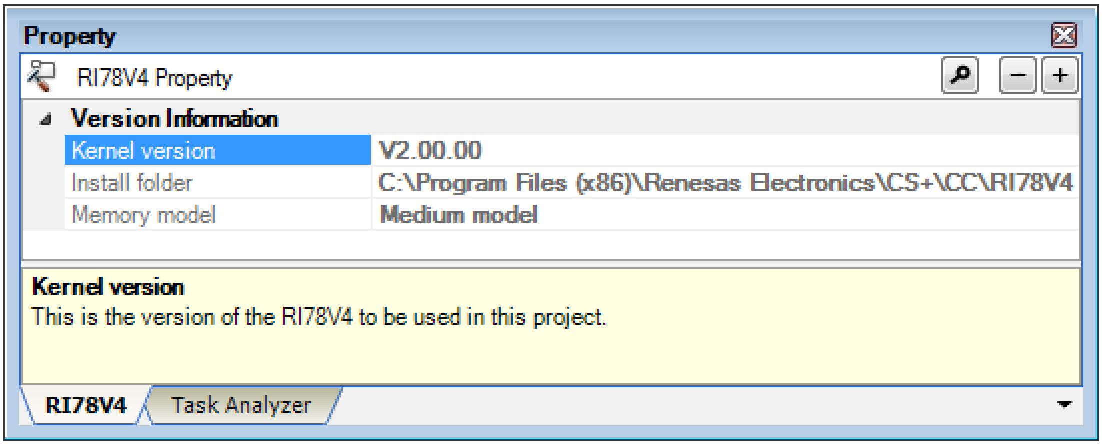
# [RI78V4] tab

## Outline

This tab shows the detailed information on the RI78V4 to be used categorized by the following.

- Version Information

## Display image




## Explanation of each area

1 ) [Version Information]

The detailed information on the version of the RI78V4 are displayed.

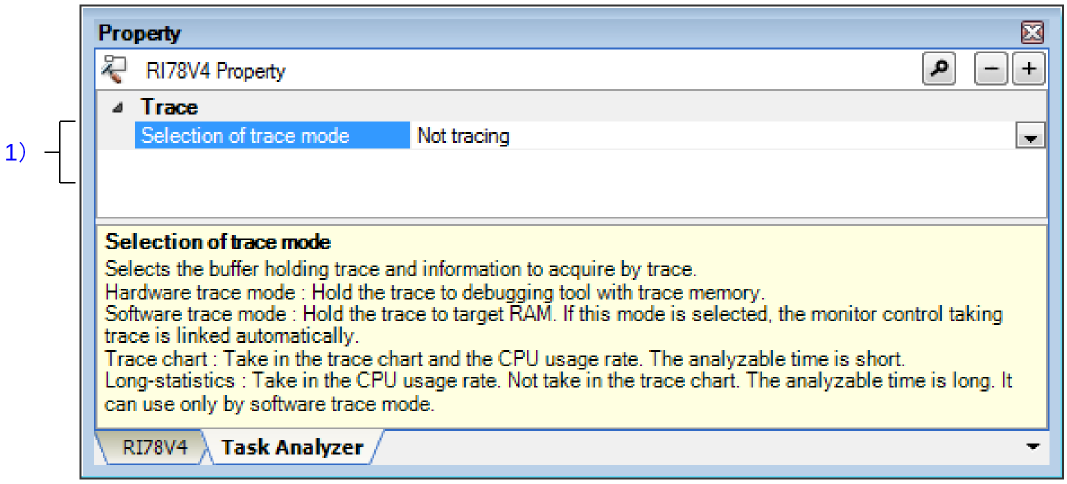
| | | |
|---|---|---|
| Kernel version | Display the version of the RI78V4 to be used.<br>Note that the version is set permanently when the project is created, and cannot be changed. | |
| | Default | *Using the RI78V4 version* |
| | How to change | Changes not allowed |
| Install folder | Display the folder in which the RI78V4 to be used is installed with the absolute path. | |
| | Default | *The folder in which the RI78V4 to be used is installed* |
| | How to change | Changes not allowed |
| Memory model | Display the memory model set in the project.<br>Display the same value as the value of the [Memory model type] property of the build tool. | |
| | Default | *The memory model selected in the property of the build tool* |
| | How to change | Changes not allowed |

# [Task Analyzer] tab

## Outline

This tab shows the detailed information on the using task analyzer in RI78V4 package.

## Display image




## How to open

- First, selects the “Real-time OS“ in Project Tree panel, after selecting a real-time OS node, selects [view] menu -> [property] or selects context menu -> [property].

Note When a property panel opens already, if you select a real-time OS node on project tree panel, detail information is displayed.

## Explanation of each area

1 ) [Trace]

Sets up the trace mode of task analyzer.

| | | |
|---|---|---|
| Selection of trace mode | Select trace mode of Realtime OS Task Analyzer | |
| | Default | Not tracing |
| | How to change | Select from the drop-down list. |

<table>
<tr><td rowspan="4"></td><td rowspan="4">Restriction</td><td>Not tracing</td><td>Can not use Realtime OS Task Analyzer.</td></tr>
<tr><td>Taking in trace chart by hardware trace mode</td><td>The trace information is collected in the trace memory which emulator or simulator has.</td></tr>
<tr><td>Taking in trace chart by software trace mode</td><td>The trace information is collected in the trace buffer secured on the user memory area.<br>To use this mode, implementation of user-own coding module and setup of the system configuration file are required.</td></tr>
<tr><td>Taking in longstatistics by software trace mode</td><td>The trace information is collected in the RI78V4's variable secured on the user memory area.<br>To use this mode, the trace buffer is allocated in “.kernel_data_trace_n“ section.</td></tr>
<tr><td rowspan="5">Operation after used up the buffers</td><td colspan="3">Select the operation after user up the trace buffer.<br>This item is displayed only when “Taking in trace chart by software trace mode” is selected.</td></tr>
<tr><td>Default</td><td colspan="2">Continue to exection while the buffers overwriting.</td></tr>
<tr><td>How to change</td><td colspan="2">Select from the drop-down list.</td></tr>
<tr><td rowspan="2">Restriction</td><td>Continue to exection while the buffers overwriting</td><td>It is overwritten sequentially from old information.</td></tr>
<tr><td>Stop the trace taking in</td><td>The RI78V4 stops tracing.</td></tr>
<tr><td rowspan="4">Buffer size</td><td colspan="3">Specify the size of the trace buffer (in bytes). Please refer to “15.4 Trace Buffer Size (Taking in Trace Chart by Software Trace Mode)” for the estimate of the size of the trace buffer.<br>This item is displayed only when “Taking in trace chart by software trace mode” is selected.</td></tr>
<tr><td>Default</td><td colspan="2">0x100</td></tr>
<tr><td>How to change</td><td colspan="2">Directly enter to the text box. Only a hexadecimal number can be entered.</td></tr>
<tr><td>Restriction</td><td colspan="2">0xa ～ 0xfffe</td></tr>
<tr><td rowspan="5">Select the buffer</td><td colspan="3">Select the buffer<br>This item is displayed only when “Taking in trace chart by software trace mode” is selected.</td></tr>
<tr><td>Default</td><td colspan="2">Kernel buffer</td></tr>
<tr><td>How to change</td><td colspan="2">Select from the drop-down list.</td></tr>
<tr><td rowspan="2">Restriction</td><td>Kernel buffer</td><td>The trace buffer is allocated in “.kernel_data_trace_n“ section.</td></tr>
<tr><td>Another buffer</td><td>The trace buffer is allocated from specified address.</td></tr>
</table>

<table>
<tr><td rowspan="4">Buffer address</td><td colspan="3">Specify the start address of the trace buffer.<br>This item is displayed only when “Another buffer” is selected.</td></tr>
<tr><td>Default</td><td colspan="2">0xf0000</td></tr>
<tr><td>How to change</td><td colspan="2">Directly enter to the text box.</td></tr>
<tr><td>Restriction</td><td colspan="2">0xf0000 ~ 0xffff4</td></tr>
<tr><td rowspan="5">Trace the timer interrupt for Real-time OS</td><td colspan="3">Specifies whether the timer interrupt is traced or not.<br>This item is displayed when “Taking in trace chart by hardware trace mode” or “Taking in trace chart by software trace mode” is selected.</td></tr>
<tr><td>Default</td><td colspan="2">Kernel buffer</td></tr>
<tr><td>How to change</td><td colspan="2">Select from the drop-down list.</td></tr>
<tr><td rowspan="2">Restriction</td><td>Yes</td><td>The timer interrupt is traced.</td></tr>
<tr><td>No</td><td>The timer interrupt is not traced.</td></tr>
</table>

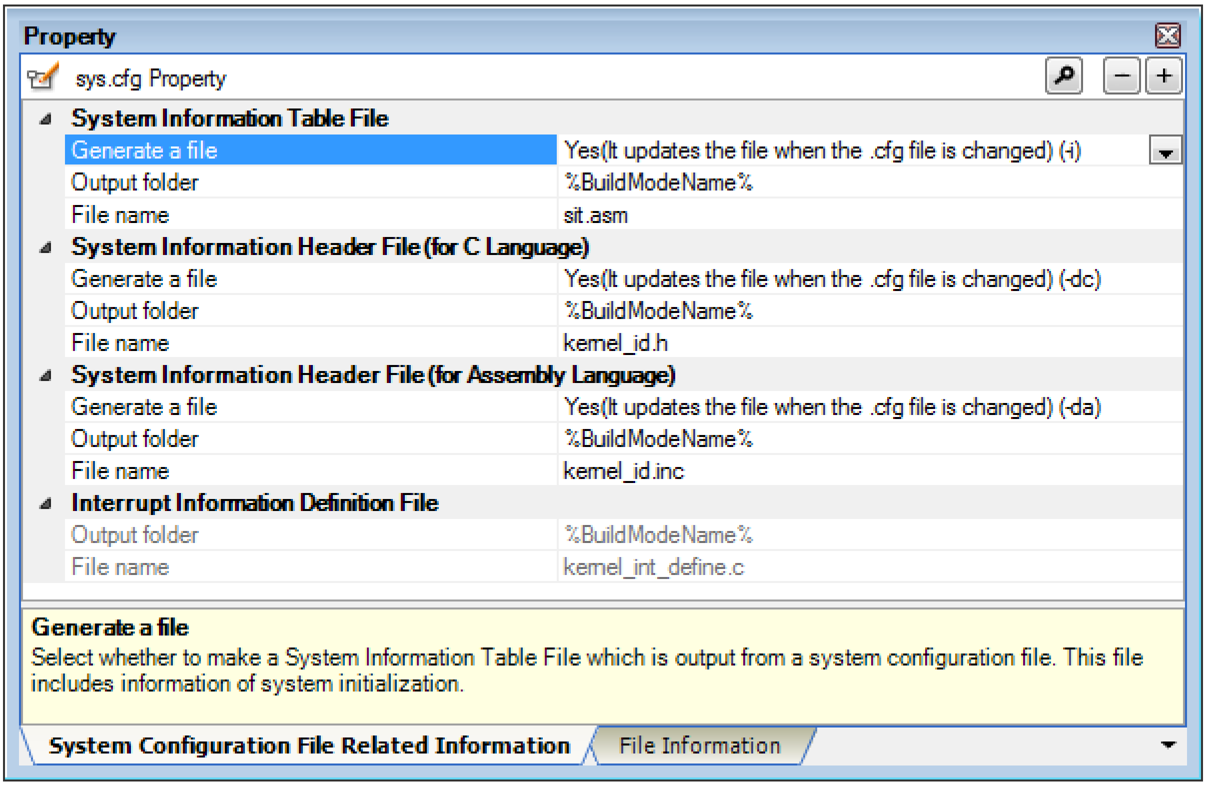
# [System Configuration File Related Information] tab

## Outline

This tab shows the detailed information on the using system configuration file categorized by the following and the configuration can be changed.

- System information table file
- System information header file (for C language)
- System information header file (for assembly language)

## Display image



## Explanation of each area

1 ) [System Information Table File]
The detailed information on the system information table file are displayed and the configuration can be changed.

<table>
<tr><td rowspan="6">Generate a file</td><td colspan="3">Select whether to generate a system information table file and whether to update the file when the system configuration file is changed.</td></tr>
<tr><td>Default</td><td colspan="2">Yes(It updates the file when the .cfg file is changed)(-i)</td></tr>
<tr><td>How to change</td><td colspan="2">Select from the drop-down list.</td></tr>
<tr><td rowspan="3">Restriction</td><td>Yes(It updates the file when the .cfg file is changed)(-i)</td><td>Generates a new system information table file and displays it on the project tree.<br>If the system configuration file is changed when there is already a system information table file, then the system information table file is updated.</td></tr>
<tr><td>Yes(It does not update the file when the .cfg file is changed)(-ni)</td><td>Does not update the system information table file when the system configuration file is changed.<br>An error occurs during build if this item is selected when the system information table file does not exist.</td></tr>
<tr><td>No(It does not register the file to the project)(-ni)</td><td>Does not generate a system information table file and does not display it on the project tree.<br>If this item is selected when there is already a system information table file, then the file itself is not deleted.</td></tr>
<tr><td rowspan="4">Output folder</td><td colspan="3">Specify the folder for outputting the system information table file.<br>If a relative path is specified, the reference point of the path is the project folder.<br>If an absolute path is specified, the reference point of the path is the project folder (unless the drives are different).<br>The following macro name is available as an embedded macro.<br>%BuildModeName%: Replaces with the build mode name.<br>If this field is left blank, macro name "%BuildModeName%" will be displayed.<br>This property is not displayed when [No(It does not register the file that is added to the project)(-ni)] in the [Generate a file] property is selected.</td></tr>
<tr><td>Default</td><td colspan="2">%BuildModeName%</td></tr>
<tr><td>How to change</td><td colspan="2">Directly enter to the text box or edit by the Browse For Folder dialog box which appears when clicking the [...] button.</td></tr>
<tr><td>Restriction</td><td colspan="2">Up to 247 characters</td></tr>
</table>

<table>
<tr><td rowspan="4">File name</td><td colspan="2">Specify the system information table file name.<br>If the file name is changed, the name of the file displayed on the project tree.<br>Use the extension ".asm". If the extension is different or omitted, ".asm" is automatically added.<br>This property is not displayed when [No(It does not register the file that is added to the project)(-ni)] in the [Generate a file] property is selected.</td></tr>
<tr><td>Default</td><td>sit.asm</td></tr>
<tr><td>How to change</td><td>Directly enter to the text box.</td></tr>
<tr><td>Restriction</td><td>Up to 259 characters</td></tr>
</table>

2 ) [System Information Header File (for C Language)]

The detailed information on the system information header file (for C language) are displayed and the configuration can be changed.

<table>
<tr><td rowspan="6">Generate a file</td><td colspan="3">Select whether to generate a system information header file (for C language) and whether to update the file when the system configuration file is changed.</td></tr>
<tr><td>Default</td><td colspan="2">Yes(It updates the file when the .cfg file is changed)(-dc)</td></tr>
<tr><td>How to change</td><td colspan="2">Select from the drop-down list.</td></tr>
<tr><td rowspan="3">Restriction</td><td>Yes(It updates the file when the .cfg file is changed)(-dc)</td><td>Generates a system information header file and displays it on the project tree.<br>If the system configuration file is changed when there is already a system information header file, then the system information header file is updated.</td></tr>
<tr><td>Yes(It does not update the file when the .cfg file is changed)(-ndc)</td><td>Does not update the system information header file when the system configuration file is changed.<br>An error occurs during build if this item is selected when the system information header file does not exist.</td></tr>
<tr><td>No(It does not register the file to the project)(-ndc)</td><td>Does not generate a system information header file and does not display it on the project tree.<br>If this item is selected when there is already a system information header file, then the file itself is not deleted.</td></tr>
<tr><td rowspan="4">Output folder</td><td colspan="3">Specify the folder for outputting the system information header file (for C language).<br>If a relative path is specified, the reference point of the path is the project folder.<br>If an absolute path is specified, the reference point of the path is the project folder (unless the drives are different).<br>The following macro name is available as an embedded macro.<br>%BuildModeName%: Replaces with the build mode name.<br>If this field is left blank, macro name "%BuildModeName%" will be displayed.<br>This property is not displayed when [No(It does not register the file that is added to the project)(-ndc)] in the [Generate a file] property is selected.</td></tr>
<tr><td>Default</td><td colspan="2">%BuildModeName%</td></tr>
<tr><td>How to change</td><td colspan="2">Directly enter to the text box or edit by the Browse For Folder dialog box which appears when clicking the [...] button.</td></tr>
<tr><td>Restriction</td><td colspan="2">Up to 247 characters</td></tr>
</table>

<table>
<tr><td rowspan="4">File name</td><td colspan="2">Specify the system information header file (for C language) name.<br>If the file name is changed, the name of the file displayed on the project tree.<br>Use the extension ".h". If the extension is different or omitted, ".h" is automatically added.<br>This property is not displayed when [No(It does not register the file that is added to the project)(-ndc)] in the [Generate a file] property is selected.</td></tr>
<tr><td>Default</td><td>kernel_id.h</td></tr>
<tr><td>How to change</td><td>Directly enter to the text box.</td></tr>
<tr><td>Restriction</td><td>Up to 259 characters</td></tr>
</table>

3 ) [System Information Header File (for Assembly Language)]

The detailed information on the system information header file (for assembly language) are displayed and the configuration can be changed.

<table>
<tr><td rowspan="6">Generate a file</td><td colspan="3">Select whether to generate a system information header file (for assembly language) and whether to update the file when the system configuration file is changed.</td></tr>
<tr><td>Default</td><td colspan="2">Yes(It updates the file when the .cfg file is changed)(-da)</td></tr>
<tr><td>How to change</td><td colspan="2">Select from the drop-down list.</td></tr>
<tr><td rowspan="3">Restriction</td><td>Yes(It updates the file when the .cfg file is changed)(-da)</td><td>Generates a system information header file and displays it on the project tree.<br>If the system configuration file is changed when there is already a system information header file, then the system information header file is updated.</td></tr>
<tr><td>Yes(It does not update the file when the .cfg file is changed)(-nda)</td><td>Does not update the system information header file when the system configuration file is changed.<br>An error occurs during build if this item is selected when the system information header file does not exist.</td></tr>
<tr><td>No(It does not register the file to the project)(-nda)</td><td>Does not generate a system information header file and does not display it on the project tree.<br>If this item is selected when there is already a system information header file, then the file itself is not deleted.</td></tr>
<tr><td rowspan="4">Output folder</td><td colspan="3">Specify the folder for outputting the system information header file (for assembly language).<br>If a relative path is specified, the reference point of the path is the project folder.<br>If an absolute path is specified, the reference point of the path is the project folder (unless the drives are different).<br>The following macro name is available as an embedded macro.<br>%BuildModeName%: Replaces with the build mode name.<br>If this field is left blank, macro name "%BuildModeName%" will be displayed.<br>This property is not displayed when [No(It does not register the file that is added to the project)(-nda)] in the [Generate a file] property is selected.</td></tr>
<tr><td>Default</td><td colspan="2">%BuildModeName%</td></tr>
<tr><td>How to change</td><td colspan="2">Directly enter to the text box or edit by the Browse For Folder dialog box which appears when clicking the [...] button.</td></tr>
<tr><td>Restriction</td><td colspan="2">Up to 247 characters</td></tr>
</table>

<table>
<tr><td rowspan="4">File name</td><td colspan="2">Specify the system information header file (for assembly language) name.<br>If the file name is changed, the name of the file displayed on the project tree.<br>Use the extension ".inc". If the extension is different or omitted, ".inc" is automatically added.<br>This property is not displayed when [No(It does not register the file that is added to the project)(-nda)] in the [Generate a file] property is selected.</td></tr>
<tr><td>Default</td><td>kernel_id.inc</td></tr>
<tr><td>How to change</td><td>Directly enter to the text box.</td></tr>
<tr><td>Restriction</td><td>Up to 259 characters</td></tr>
</table>

4 ) [Interrupt Information Definition File]

The detailed information on the interrupt information definition file are displayed.

| | | |
|---|---|---|
| Output folder | Display the folder for outputting the interrupt information definition file. | |
| | Default | %BuildModeName% |
| | How to change | Changes not allowed. |
| | Restriction | Up to 247 characters |
| File name | Display the interrupt information definition file. | |
| | Default | .kernel_int_define.c |
| | How to change | Changes not allowed. |

# APPENDIX B CAUTIONS

## B.1 Restriction of Compiler Option

Systems embedding the RI78V4 cannot use the following compile options.

| Option | Meaning |
|---|---|
| -base_number=prefix | This option specifies the notation of the radix for numeric constants. Specifies the prefix notation (0xn...n). |

## B.2 Handling Register Bank

Systems embedding the RI78V4 should generally operate with register bank 0.

If it is necessary to change the register bank, do so in accordance with the specifications below. Changing the register bank is enabled for some routines, and disabled for others.

[Routines where changing the register bank is enabled]

- Task
  In the task, the initial register bank number is set permanently to 0.
  When switching tasks in the RI78V4, only the register bank number and one bank's worth of general registers (task-switching bank) are retired/restored.
  The remaining three banks of general registers are not retired or restored, so if more than two register banks are to be used in the task process, then when changing the register banks, the general register of the register bank before the change must be retired. If it is not retired, then the register bank could be corrupted in the task that is switched to.

- Interrupt servicing not managed by an OS
  When changing a register bank in an interrupt process not matched by the OS, restore the register bank number of the interrupt source when the interrupt ends.

[Routines where changing the register bank is disabled]

- Interrupt handler
  Interrupt handlers inherit the register bank number of the source of the interrupt.

- Cyclic handler
  Cyclic handlers inherit the register bank number of the source of the timer handler interrupt.

- Idle routine
  In the idle routine, the initial register bank number is set permanently to 0.

- Initialization routine
  In the initialization routine, the initial register bank number is set permanently to 0. It is overwritten by register bank 0, regardless of the register bank set before OS initialization (before the call to the __urx_start function).

Revision Record

| Rev. | Date | Description | |
|---|---|---|---|
| | | Page | Summary |
| 1.00 | Mar 25, 2015 | - | First Edition issued |

RI78V4 V2.00.00 User's Manual:
Coding

Publication Date: Rev.1.00 Mar 25, 2015

Published by: Renesas Electronics Corporation

**SALES OFFICES** Renesas Electronics Corporation http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.


**Renesas Electronics America Inc.**
2801 Scott Boulevard Santa Clara, CA 95050-2549, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

**Renesas Electronics Canada Limited**
9251 Yonge Street, Suite 8309 Richmond Hill, Ontario Canada L4C 9T3
Tel: +1-905-237-2004

**Renesas Electronics Europe Limited**
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-585-100, Fax: +44-1628-585-900

**Renesas Electronics Europe GmbH**
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-6503-0, Fax: +49-211-6503-1327

**Renesas Electronics (China) Co., Ltd.**
Room 1709, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100191, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

**Renesas Electronics (Shanghai) Co., Ltd.**
Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, P. R. China 200333
Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

**Renesas Electronics Hong Kong Limited**
Unit 1601-1611, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2265-6688, Fax: +852 2886-9022

**Renesas Electronics Taiwan Co., Ltd.**
13F, No. 363, Fu Shing North Road, Taipei 10543, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

**Renesas Electronics Singapore Pte. Ltd.**
80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre, Singapore 339949
Tel: +65-6213-0200, Fax: +65-6213-0300

**Renesas Electronics Malaysia Sdn.Bhd.**
Unit 1207, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

**Renesas Electronics India Pvt. Ltd.**
No.777C, 100 Feet Road, HALII Stage, Indiranagar, Bangalore, India
Tel: +91-80-67208700, Fax: +91-80-67208777

**Renesas Electronics Korea Co., Ltd.**
12F., 234 Teheran-ro, Gangnam-Gu, Seoul, 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141





Colophon 4.0

RI78V4 V2.00.00

Renesas Electronics Corporation

R20UT3375EJ0100