User's Manual

RI600V4

Real-Time Operating System

User's Manual: Coding

Target Device

RX Family

All information contained in these materials, including products and product specifications, represents information on the product at the time of publication and is subject to change by Renesas Electronics Corp. without notice. Please review the latest information published by Renesas Electronics Corp. through various means, including the Renesas Electronics Corp. website (http://www.renesas.com).

Renesas Electronics
www.renesas.com

Rev.1.04 Sep 2013

How to Use This Manual

Readers This manual is intended for users who design and develop application system using RX family.

Purpose This manual is intended for users to understand the functions of real-time OS “RI600V4” manufactured by Renesas Electronics, described the organization listed below.

Organization This manual can be broadly divided into the following units.

CHAPTER 1 OVERVIEW
CHAPTER 2 SYSTEM BUILDING
CHAPTER 3 TASK MANAGEMENT FUNCTIONS
CHAPTER 4 TASK DEPENDENT SYNCHRONIZATION FUNCTIONS
CHAPTER 5 SYNCHRONIZATION AND COMMUNICATION FUNCTIONS
CHAPTER 6 EXTENDED SYNCHRONIZATION AND COMMUNICATION FUNCTIONS
CHAPTER 7 MEMORY POOL MANAGEMENT FUNCTIONS
CHAPTER 8 TIME MANAGEMENT FUNCTIONS
CHAPTER 9 SYSTEM STATE MANAGEMENT FUNCTIONS
CHAPTER 10 INTERRUPT MANAGEMENT FUNCTIONS
CHAPTER 11 SYSTEM CONFIGURATION MANAGEMENT FUNCTIONS
CHAPTER 12 OBJECT RESET FUNCTIONS
CHAPTER 13 SYSTEM DOWN
CHAPTER 14 SCHEDULING FUNCTION
CHAPTER 16 SYSTEM INITIALIZATION
CHAPTER 17 DATA TYPES AND MACROS
CHAPTER 18 SERVICE CALLS
CHAPTER 19 SYSTEM CONFIGURATION FILE
CHAPTER 20 CONFIGURATOR cfg600
CHAPTER 21 TABLE GENARATION UTILITY mkritbl
APPENDIX A WINDOW REFERENCE
APPENDIX B FLOATING-POINT OPERATION FUNCTION
APPENDIX C DSP FUNCTION
APPENDIX D STACK SIZE ESTIMATION

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, microcomputers, C language, and assemblers.

To understand the hardware functions of the RX MCU
→ 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) $2^{20} = 1024^2$

up4(*data*):	A value in which *data* is rounded up to the multiple of 4.
down(*data*):	A integer part of *data*.

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
RI600V4	Coding	This document
	Debug	R20UT0775E
	Analysis	R20UT2185E

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

TABLE OF CONTENTS

CHAPTER 1 OVERVIEW 12

1.1 Outline 12
1.1.1 Real-time OS 12
1.1.2 Multi-task OS 12

CHAPTER 2 SYSTEM BUILDING 13

2.1 Outline 13
2.2 Coding Processing Programs 14
2.3 Coding System Configuration File 14
2.4 Coding User-Own Coding Module 15
2.5 Creating Load Module 16
2.6 Build Options 21
2.6.1 Service call information files and "-ri600_preinit_mrc" compiler option 21
2.6.2 Compiler option for the boot processing file 22
2.6.3 Kernel library 23
2.6.4 Arrangement of section 24
2.6.5 Initialized data section 25
2.6.6 Options for Realtime OS Task Analyzer 26

CHAPTER 3 TASK MANAGEMENT FUNCTIONS 27

3.1 Outline 27
3.2 Tasks 27
3.2.1 Task state 27
3.2.2 Task priority 29
3.2.3 Basic form of tasks 30
3.2.4 Internal processing of task 31
3.2.5 Processor mode of task 31
3.3 Create Task 32
3.4 Activate Task 32
3.4.1 Activate task with queuing 32
3.4.2 Activate task without queuing 33
3.5 Cancel Task Activation Requests 34
3.6 Terminate Task 35
3.6.1 Terminate invoking task 35
3.6.2 Terminate Another task 36
3.7 Change Task Priority 37
3.8 Reference Task Priority 38
3.9 Reference Task State 39
3.9.1 Reference task state 39
3.9.2 Reference task state (simplified version) 40

CHAPTER 4 TASK DEPENDENT SYNCHRONIZATION FUNCTIONS 41

4.1 Outline 41
4.2 Put Task to Sleep 41
4.2.1 Waiting forever 41
4.2.2 With time-out 42
4.3 Wake-up Task 43
4.4 Cancel Task Wake-up Requests 44
4.5 Forcibly Release Task from Waiting 45
4.6 Suspend Task 46
4.7 Resume Suspended Task 47
4.7.1 Resume suspended task 47
4.7.2 Forcibly resume suspended task 48
4.8 Delay Task 49
4.9 Differences Between Sleep with Time-out and Delay 50

CHAPTER 5 SYNCHRONIZATION AND COMMUNICATION FUNCTIONS 51

5.1 Outline 51
5.2 Semaphores 51
5.2.1 Create semaphore 51
5.2.2 Acquire semaphore resource 52
5.2.3 Release semaphore resource 55
5.2.4 Reference semaphore state 56
5.3 Eventflags 57
5.3.1 Create eventflag 57
5.3.2 Set eventflag 58
5.3.3 Clear eventflag 59
5.3.4 Check bit pattern 60
5.3.5 Reference eventflag state 65
5.4 Data Queues 66
5.4.1 Create data queue 66
5.4.2 Send to data queue 67
5.4.3 Forced send to data queue 72
5.4.4 Receive from data queue 73
5.4.5 Reference data queue state 78
5.5 Mailboxes 79
5.5.1 Messages 79
5.5.2 Create mailbox 81
5.5.3 Send to mailbox 81
5.5.4 Receive from mailbox 82
5.5.5 Reference mailbox state 85

CHAPTER 6 EXTENDED SYNCHRONIZATION AND COMMUNICATION FUNCTIONS 86

6.1 Outline 86
6.2 Mutexes 86
6.2.1 Priority inversion problem 87
6.2.2 Current priority and base priority 87
6.2.3 Simplified priority ceiling protocol 87

6.2.4 Differences from semaphores 88
6.2.5 Create mutex 88
6.2.6 Lock mutex 89
6.2.7 Unlock mutex 92
6.2.8 Reference mutex state 93
6.3 Message Buffers 94
6.3.1 Create message buffer 94
6.3.2 Send to message buffer 95
6.3.3 Receive from message buffer 100
6.3.4 Reference message buffer state 105

CHAPTER 7 MEMORY POOL MANAGEMENT FUNCTIONS 106

7.1 Outline 106
7.2 Fixed-Sized Memory Pools 107
7.2.1 Create fixed-sized memory pool 107
7.2.2 Acquire fixed-sized memory block 108
7.2.3 Release fixed-sized memory block 113
7.2.4 Reference fixed-sized memory pool state 114
7.3 Variable-Sized Memory Pools 115
7.3.1 Create variable-sized memory pool 115
7.3.2 Size of Variable-sized memory block. 116
7.3.3 Acquire variable-sized memory block 117
7.3.4 Release variable-sized memory block 122
7.3.5 Reference variable-sized memory pool state 123

CHAPTER 8 TIME MANAGEMENT FUNCTIONS 124

8.1 Outline 124
8.2 System Time 124
8.2.1 Base clock timer interrupt 124
8.2.2 Base clock interval 124
8.3 Timer Operations 125
8.4 Delay task 125
8.5 Time-out 125
8.6 Cyclic handlers 126
8.6.1 Basic form of cyclic handlers 126
8.6.2 Processing in cyclic handler 127
8.6.3 Create cyclic handler 127
8.6.4 Start cyclic handler operation 128
8.6.5 Stop cyclic handler operation 130
8.6.6 Reference cyclic handler state 131
8.7 Alarm Handlers 132
8.7.1 Basic form of alarm handler 132
8.7.2 Processing in alarm handler 133
8.7.3 Create alarm handler 133
8.7.4 Start alarm handler operation 134
8.7.5 Stop alarm handler operation 135
8.7.6 Reference alarm handler state 136

8.8 System Time . . . 137
8.8.1 Set system time . . . 137
8.8.2 Reference system time . . . 138
8.9 Initialize Base Clock Timer . . . 139

CHAPTER 9 SYSTEM STATE MANAGEMENT FUNCTIONS . . . 140

9.1 Outline . . . 140
9.2 Rotate Task Precedence . . . 140
9.3 Reference Task ID in the RUNNING State . . . 142
9.4 Lock and Unlock the CPU . . . 143
9.5 Reference CPU Locked State . . . 145
9.6 Disable and Enable Dispatching . . . 146
9.7 Reference Dispatching State . . . 147
9.8 Reference Context Type . . . 148
9.9 Reference Dispatch Pending State . . . 149

CHAPTER 10 INTERRUPT MANAGEMENT FUNCTIONS . . . 150

10.1 Interrupt Type . . . 150
10.2 Fast Interrupt of the RX-MCU . . . 150
10.3 CPU Exception . . . 150
10.4 Base Clock Timer Interrupt . . . 151
10.5 Multiple Interrupts . . . 151
10.6 Interrupt Handlers . . . 152
10.6.1 Basic form of interrupt handlers . . . 152
10.6.2 Register interrupt handler . . . 153
10.7 Maskable Interrupt Acknowledgement Status in Processing Programs . . . 153
10.8 Prohibit Maskable Interrupts . . . 154
10.8.1 Move to the CPU locked state by using loc_cpu, iloc_cpu . . . 154
10.8.2 Change PSW.IPL by using chg_ims, ichg_ims . . . 154
10.8.3 Change PSW.I and PSW.IPL directly (only for handlers) . . . 154

CHAPTER 11 SYSTEM CONFIGURATION MANAGEMENT FUNCTIONS . . . 155

11.1 Outline . . . 155
11.2 Reference Version Information . . . 155

CHAPTER 12 OBJECT RESET FUNCTIONS . . . 156

12.1 Outline . . . 156
12.2 Reset Data Queue . . . 156
12.3 Reset Mailbox . . . 157
12.4 Reset Message Buffer . . . 158
12.5 Reset Fixed-sized Memory Pool . . . 159
12.6 Reset Variable-sized Memory Pool . . . 160

CHAPTER 13 SYSTEM DOWN . . . 161

13.1 Outline 161
13.2 User-Own Coding Module 161
13.2.1 System down routine (_RI_sys_dwn__) 161
13.2.2 Parameters of system down routine 162

CHAPTER 14 SCHEDULING FUNCTION 164

14.1 Outline 164
14.2 Processing Unit and Precedence 164
14.3 Task Drive Method 164
14.4 Task Scheduling Method 165
14.4.1 Ready queue 165
14.5 Task Scheduling Lock Function 166
14.6 Idling 167
14.7 Task Scheduling in Non-Tasks 167

CHAPTER 15 REALTIME OS TASK ANALYZER 168

15.1 Outline 168
15.2 Trace Mode 168
15.3 User-Own Coding Module for Software Trace Mode 170
15.3.1 Taking in trace chart by software trace mode 170
15.3.2 Taking in long-statistics by software trace mode 173
15.4 Trace Buffer Size (Taking in Trace Chart by Software Trace Mode) 176
15.5 Error of Total Execution Time 176

CHAPTER 16 SYSTEM INITIALIZATION 177

16.1 Outline 177
16.2 Boot Processing File (User-Own Coding Module) 178
16.2.1 Boot processing function (PowerON_Reset_PC( )) 178
16.2.2 Include kernel_ram.h and kernel_rom.h 179
16.2.3 Compiler option for boot processing file 179
16.2.4 Example of the boot processing file 180
16.3 Kernel Initialization Module (vsta_knl, ivsta_knl) 183
16.4 Section Initialization Function (_INITSCT( )) 184
16.4.1 Section information file (User-Own Coding Module) 184
16.5 Registers in Fixed Vector Table/Exception Vector table 185

CHAPTER 17 DATA TYPES AND MACROS 186

17.1 Data Types 186
17.2 Macros 188
17.2.1 Constant macros 188
17.2.2 Function Macros 191

CHAPTER 18 SERVICE CALLS 192

18.1 Outline 192
18.1.1 Method for calling service calls 193
18.2 Explanation of Service Call 194
18.2.1 Task management functions 196
18.2.2 Task dependent synchronization functions 215
18.2.3 Synchronization and communication functions (semaphores) 232
18.2.4 Synchronization and communication functions (eventflags) 242
18.2.5 Synchronization and communication functions (data queues) 253
18.2.6 Synchronization and communication functions (mailboxes) 270
18.2.7 Extended synchronization and communication functions (mutexes) 281
18.2.8 Extended synchronization and communication functions (message buffers) 291
18.2.9 Memory pool management functions (fixed-sized memory pools) 306
18.2.10 Memory pool management functions (variable-sized memory pools) 317
18.2.11 Time management functions 329
18.2.12 System state management functions 342
18.2.13 Interrupt management functions 358
18.2.14 System configuration management functions 362
18.2.15 Object reset functions 365

CHAPTER 19 SYSTEM CONFIGURATION FILE 371

19.1 Outline 371
19.2 Default System Configuration File 372
19.3 Configuration Information (static API) 372
19.4 System Information (system) 373
19.5 Note Concerning system.context 376
19.5.1 Note concerning FPU and DSP 376
19.5.2 Relationship with the compiler options "-fint_register", "-base" and "-pid" . . 378
19.6 Base Clock Interrupt Information (clock) 379
19.7 Task Information (task[]) 381
19.8 Semaphore Information (semaphore[]) 384
19.9 Eventflag Information (flag[]) 386
19.10 Data Queue Information (dataqueue[]) 388
19.11 Mailbox Information (mailbox[]) 390
19.12 Mutex Information (mutex[]) 392
19.13 Message Buffer Information (message_buffer[]) 393
19.14 Fixed-sized Memory Pool Information (memorypool[]) 395
19.15 Variable-sized Memory Pool Information (variable_memorypool[]) 397
19.16 Cyclic Handler Information (cyclic_hand[]) 399
19.17 Alarm Handler Information (alarm_handl[]) 402
19.18 Relocatable Vector Information (interrupt_vector[]) 404
19.19 Fixed Vector/Exception Vector Information (interrupt_fvector[]) 407
19.20 RAM Capacity Estimation 410
19.20.1 BRI_RAM section 411
19.20.2 BRI_HEAP section 413
19.20.3 SURI_STACK section 413
19.20.4 SI section 413
19.21 Description Examples 414

CHAPTER 20 CONFIGURATOR cfg600 416

20.1 Outline 416

20.2 Start cfg600 . . . 417
20.2.1 Start cfg600 from command line . . . 417
20.2.2 Start cfg600 from CubeSuite+ . . . 417

CHAPTER 21 TABLE GENARATION UTILITY mkritbl . . . 418

21.1 Outline . . . 418
21.2 Start mkritbl . . . 419
21.2.1 Start mkritbl from command line . . . 419
21.2.2 Start mkritbl from CubeSuite+ . . . 419
21.3 Notes . . . 419

APPENDIX A WINDOW REFERENCE . . . 420

A.1 Description . . . 420

APPENDIX B FLOATING-POINT OPERATION FUNCTION . . . 436

B.1 When Using Floating-point Arithmetic Instructions in Tasks . . . 436
B.2 When Using Floating-point Arithmetic Instructions in Handlers . . . 436

APPENDIX C DSP FUNCTION . . . 437

C.1 When Using DSP Instructions in Tasks . . . 437
C.2 When Using DSP Instructions in Handlers . . . 437

APPENDIX D STACK SIZE ESTIMATION . . . 438

D.1 Types of Stack . . . 438
D.2 Call Walker . . . 438
D.3 User Stack Size Estimation . . . 439
D.4 System Stack Size Estimation . . . 440

CHAPTER 1 OVERVIEW

1.1 Outline

The RI600V4 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 RI600V4 is a high-speed, compact OS capable of being stored in and run from the ROM of a target system.
The RI600V4 is based on the μITRON4.0 specification.

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.

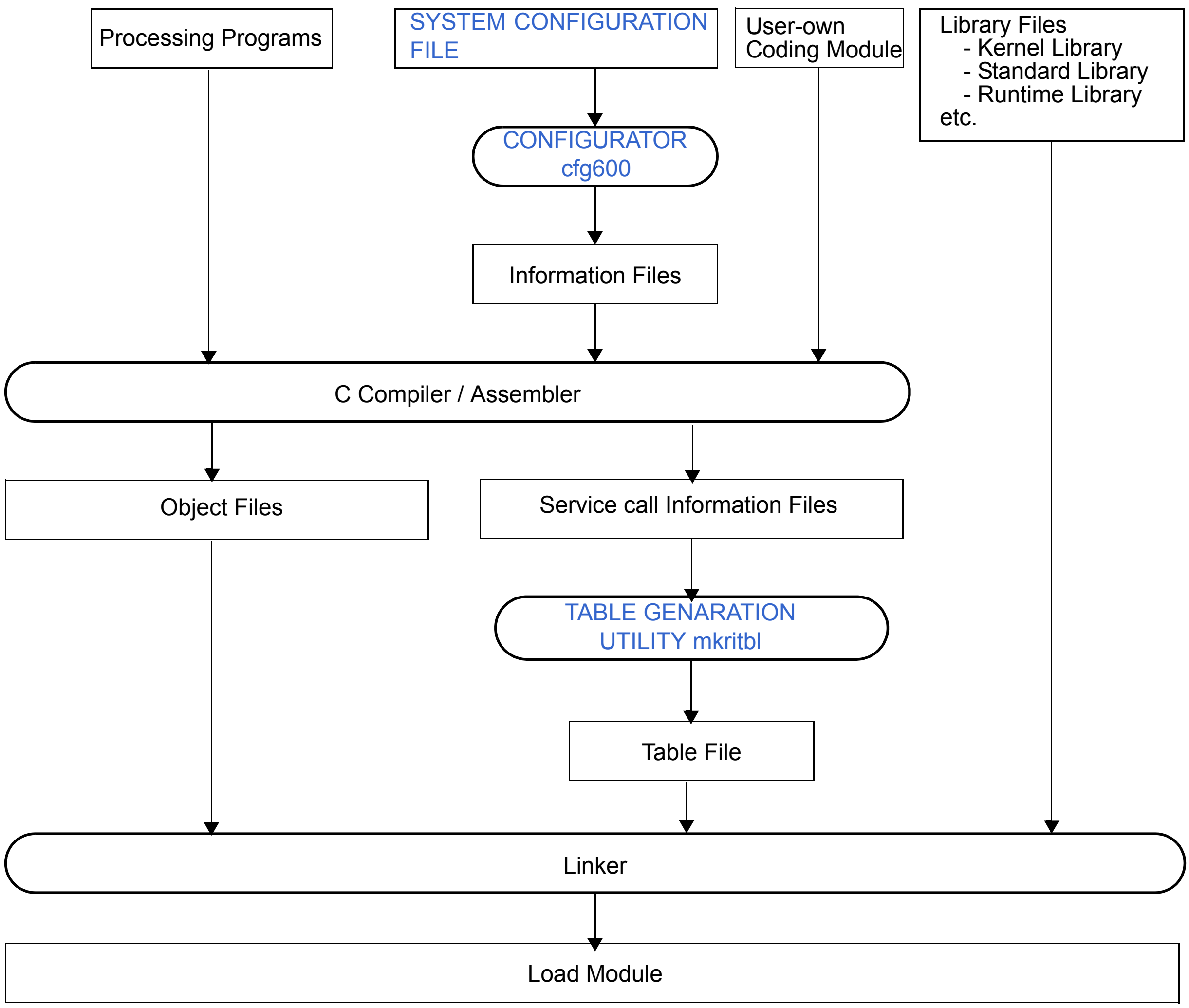
CHAPTER 2 SYSTEM BUILDING

This chapter describes how to build a system (load module) that uses the functions provided by the RI600V4.

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 RI600V4's supply media.
Figure 2-1 shows the procedure of system building

Figure 2-1 Example of System Building



The RI600V4 provides a sample program with the files necessary for generating a load module.
The sample programs are stored in the following folder. The source files are stored in “appli” sub-folder.

<ri_sample> = <CubeSuite+_root>\SampleProjects\RX*device_name*_RI600V4

- <CubeSuite+_root>
  Indicates the installation folder of CubeSuite+.
  The default folder is “C:\Program Files\Renesas Electronics\CubeSuite+\”.

- SampleProjects
  Indicates the sample project folder of CubeSuite+.

- RX
  Indicates the sample project folder of RX MCU.

- *device_name*_RI600V4
  Indicates the sample project folder of the RI600V4. The project file is stored in this folder.

  device_name: Indicates the device name which the sample is provided.

2.2 Coding Processing Programs

Code the processing that should be implemented in the system.
In the RI600V4, the processing program is classified into the following four 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 RI600V4, unlike other processing programs (interrupt handler, cyclic handler and alarm handler).

- Cyclic handlers
  The cyclic handler is a routine started for every specified cycle time.
  The RI600V4 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.

- Alarm Handlers
  The alarm handler is a routine started only once after the specified time.
  The RI600V4 handles the alarm 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.

- Interrupt Handlers
  The interrupt handler is a routine started when an interrupt occurs.
  The RI600V4 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 For details about the processing programs, refer to “CHAPTER 3 TASK MANAGEMENT FUNCTIONS”, “CHAPTER 8 TIME MANAGEMENT FUNCTIONS”, “CHAPTER 10 INTERRUPT MANAGEMENT FUNCTIONS”.

2.3 Coding System Configuration File

Code the SYSTEM CONFIGURATION FILE required for creating information files that contain data to be provided for the RI600V4.

Note 1 For details about the system configuration file, refer to “CHAPTER 19 SYSTEM CONFIGURATION FILE”.

Note 2 When the Realtime OS Task analyzer is used in “Taking in trace chart by software trace mode” or “Taking in long-statistics by software trace mode”, it is necessary to define the interrupt handler implemented in user-own coding module to the system configuration file. For details, refer to “CHAPTER 15 REALTIME OS TASK ANALYZER”.

2.4 Coding User-Own Coding Module

- SYSTEM DOWN
  - System down routine (_RI_sys_dwn__)
    The system down routine is called when the system down occurs.
- REALTIME OS TASK ANALYZER
  - User-Own Coding Module for Software Trace Mode
    When using the software trace mode, user-own coding module to get time-stamp must be implemented.
- SYSTEM INITIALIZATION
  - Boot processing function (PowerON_Reset_PC( ))
    The boot processing is defined in the reset vector, and dedicated to initialization processing that is extracted as a user-own coding module to initialize the minimum required hardware for the RI600V4 to perform processing.
    And the boot processing plays the role to take the ROM definition file and RAM definition file which are generated by the cfg600.
  - Section information file (User-Own Coding Module)
    Informations for uninitialized data sections and initialized data sections are defined in the section information file.

Note For details about the user-own coding module, refer to "CHAPTER 13 SYSTEM DOWN", "CHAPTER 15 REALTIME OS TASK ANALYZER" and "CHAPTER 16 SYSTEM INITIALIZATION".

2.5 Creating Load Module

Run a build on CubeSuite+ for files created in sections from “2.2 Coding Processing Programs” to “2.4 Coding User-Own Coding Module”, and library files provided by the RI600V4 and C compiler package, to create a load module.

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”, “CubeSuite+ Integrated Development Environment User's Manual: Start” and the Release Notes of this product for details about creating a new project or loading an existing one.

2) Set a build target project

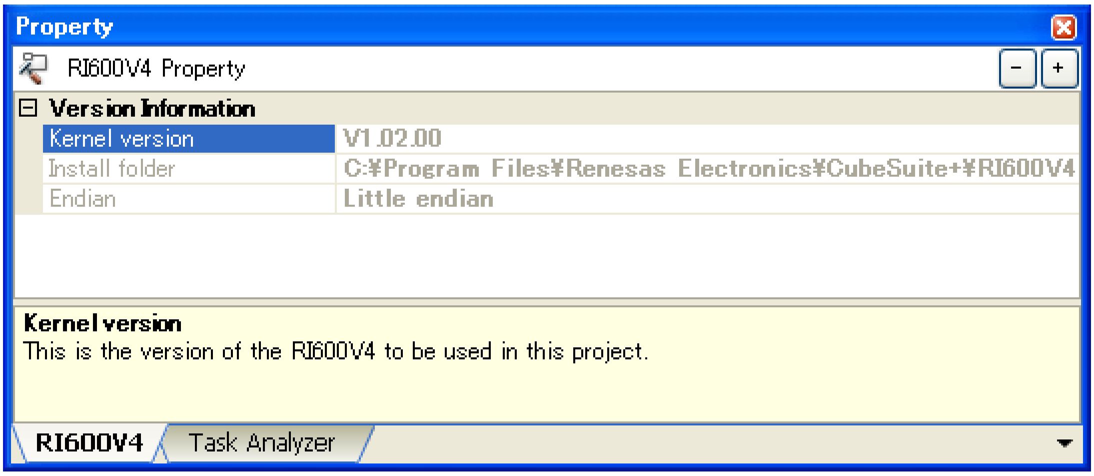
When making settings for or running a build, set the active project.
If there is no subproject, the project is always active.

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

3) Confirm the version

Select the Realtime OS node on the project tree to open the Property panel.
Confirm the version of RI600V4 to be used in the [Kernel version] property on the [RI600V4] tab.

Figure 2-2 Property Panel: [RI600V4] Tab



4) Set build target files

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

Note See “CubeSuite+ Integrated Development Environment User's Manual: RX 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.

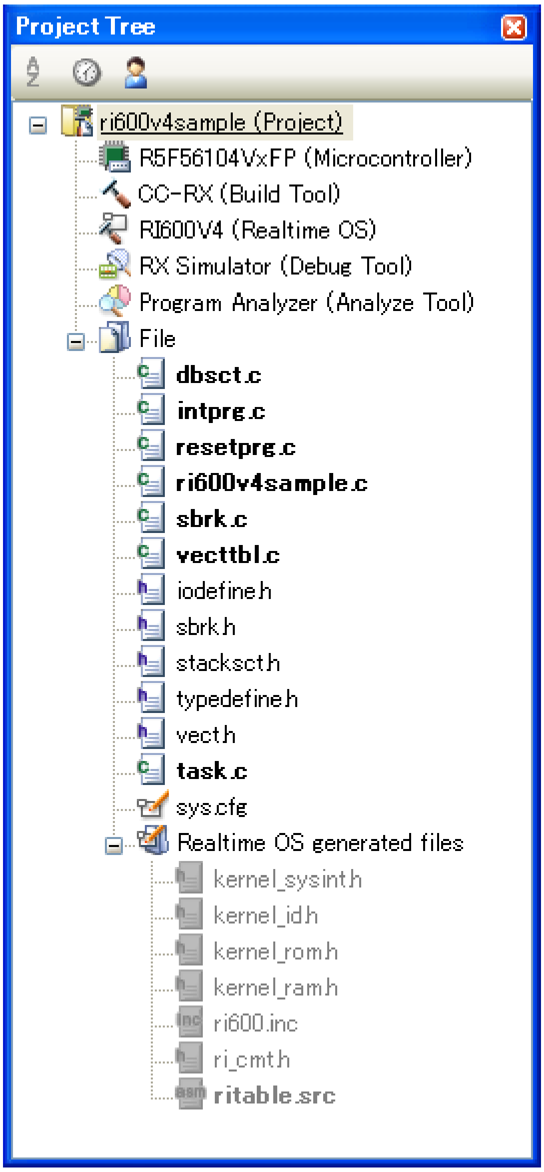
- Source files created in “2.2 Coding Processing Programs”
  - Processing programs (tasks, cyclic handlers, alarm handlers, interrupt handlers)
- System configuration file created in “2.3 Coding System Configuration File”
  - SYSTEM CONFIGURATION FILE

  Note Specify “cfg” as the extension 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”).
- Source files created in “2.4 Coding User-Own Coding Module”
  - User-own coding module (system down routine, boot processing)
- Library files provided by the RI600V4
  - Kernel library (refer to “2.6.3 Kernel library”)
- Library files provided by the C compiler 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 header file (kernel_id.h)
- Service call definition file (kernel_sysint.h)
- ROM definition file (kernel_rom.h)
- RAM definition file (kernel_ram.h)
- System definition file (ri600.inc)
- CMT timer definition file (ri_cmt.h)
- Table file (ritable.src)

Figure 2-3 Project Tree Panel



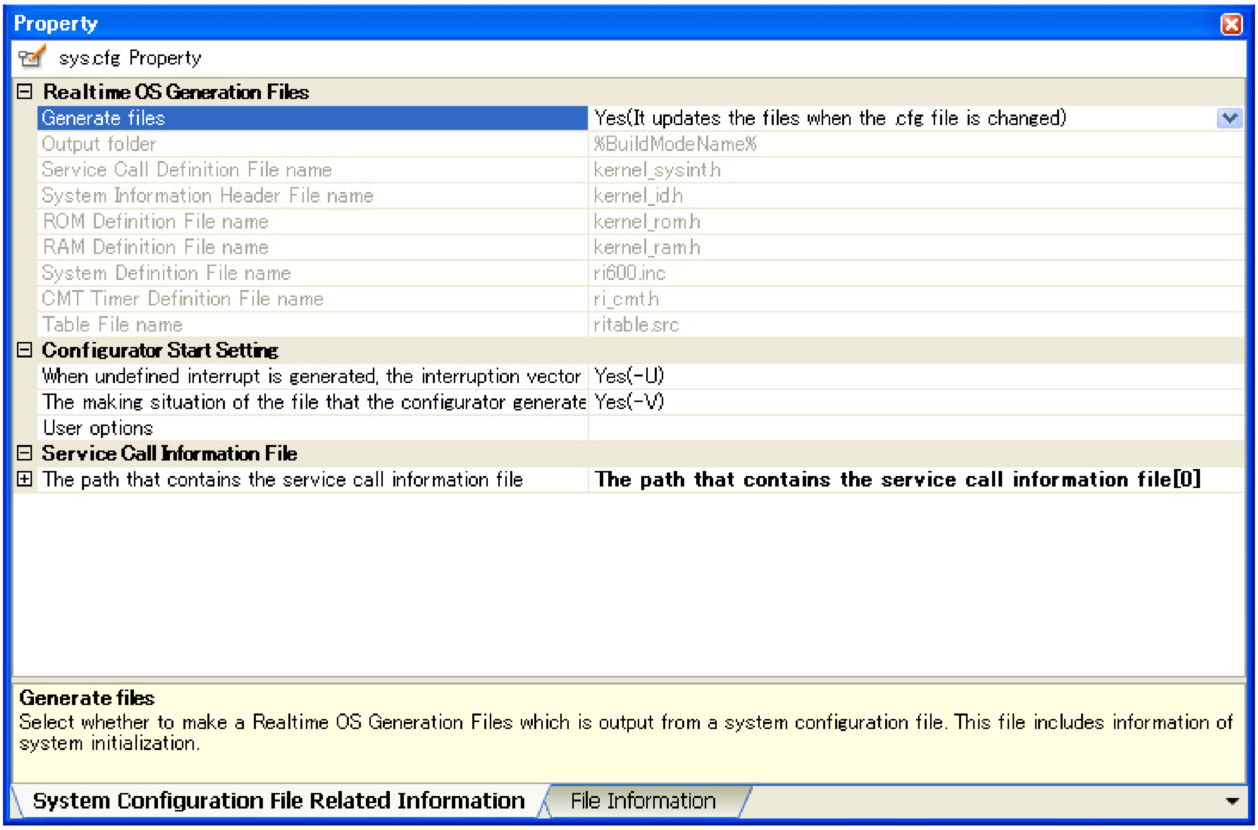
Note 2 When replacing the system configuration file, first remove the added system configuration file from the project, then add another one again.

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.

5) Set the output of Realtime OS generation 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 realtime OS generation files, etc.

Figure 2-4 Property Panel: [System Configuration File Related Information] Tab



6) Specify the output of a load module file

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

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

7) Set build options

Set the options for the compiler, assembler, linker, and the like.
Please be sure to refer to “2.6 Build Options”.

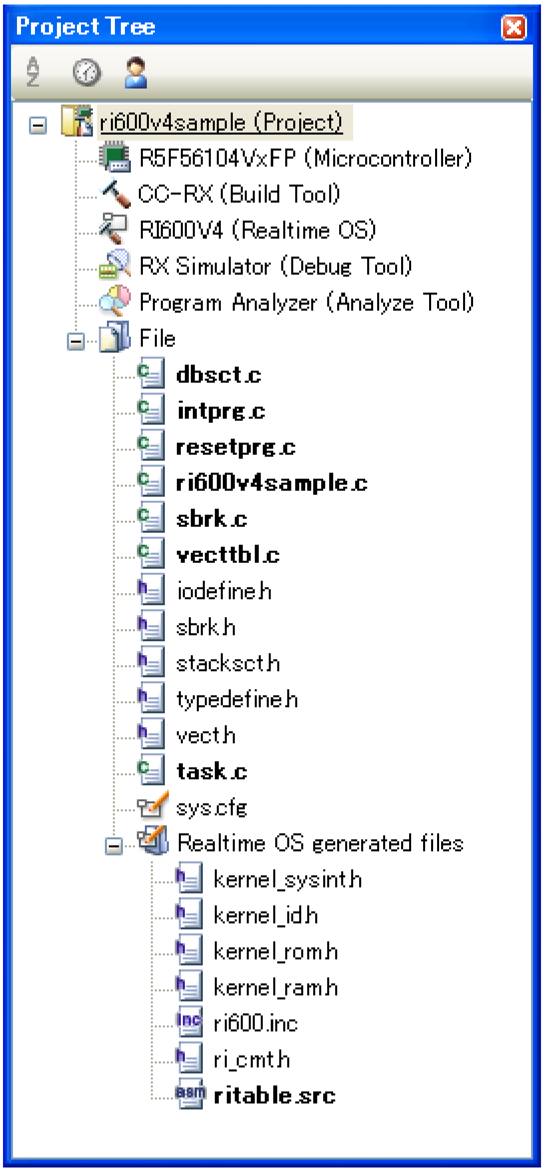
Note See “CubeSuite+ Integrated Development Environment User's Manual: RX Build” for details about setting build options.

8) Run a build

Run a build to create a load module.

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

2.6 Build Options

This section explains the build options that should be especially noted.

2.6.1 Service call information files and “-ri600_preinit_mrc” compiler option

The service call information file (mrc files) are generated to the same folder as object files at compilation of the source files that includes kernel.h file.
The name of service calls used in the source files are outputted in the mrc files. It is necessary to input all files to the table generation utility mkritbl. If there is a leaking in the input file, service call modules that application uses might not be linked. In this case, the system down will occur when the service call is issued.
On the other hand, if the mrc files which are generated in the past and which is invalid in now are input to the mkritbl, the service call modules that are not used in the application may be linked. In this case, there is no problem in the operation of the RI600V4 but the module size uselessly grows.
Specify “-ri600_preinit_mrc” compiler option for the source file that includes kernel.h file even if this option is not specified, there is no problem in the operation of the RI600V4 but the service call module that is not used in the application may be linked.
When application libraries are used, the mrc files that is generated at compilation of the library source should be inputted to the mkritbl. If this way is difficult for you, make mrc file where name of using service calls is described (see belows), and input the mrc file to the mkritbl.
Note, the system down will occur when the service call that is not linked is called.

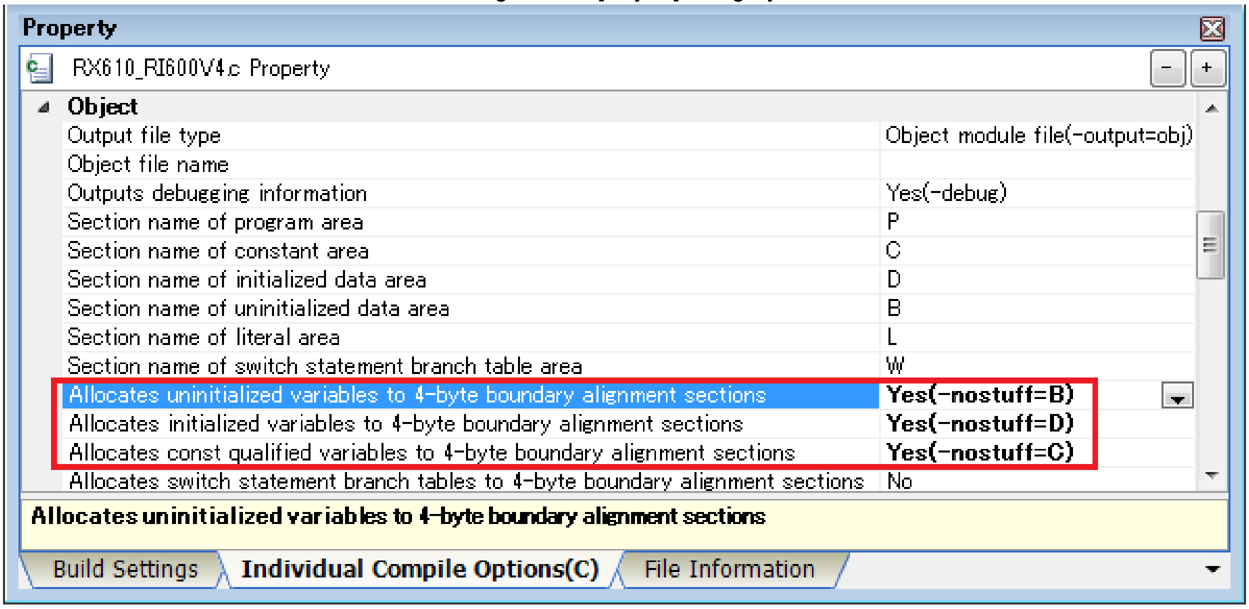
```
sta_tsk
snd_mbx
rcv_mbx
prcv_mbx
```

2.6.2 Compiler option for the boot processing file

It is necessary to set “-nostuff” option for the boot processing file (“resetprg.c” in the sample project) like a mention in “16.2.3 Compiler option for boot processing file”. If not, the RI600V4 does not work correctly.
To set “-nostuff” option only for the boot processing file, please set any of the following in the [Individual Compile Options] tab of [Property] panel for the boot processing file. To set “-nostuff” option for all, please set any of the following in the [Compiler Options] tab of [Property] panel for [CC-RX (Build Tool)].

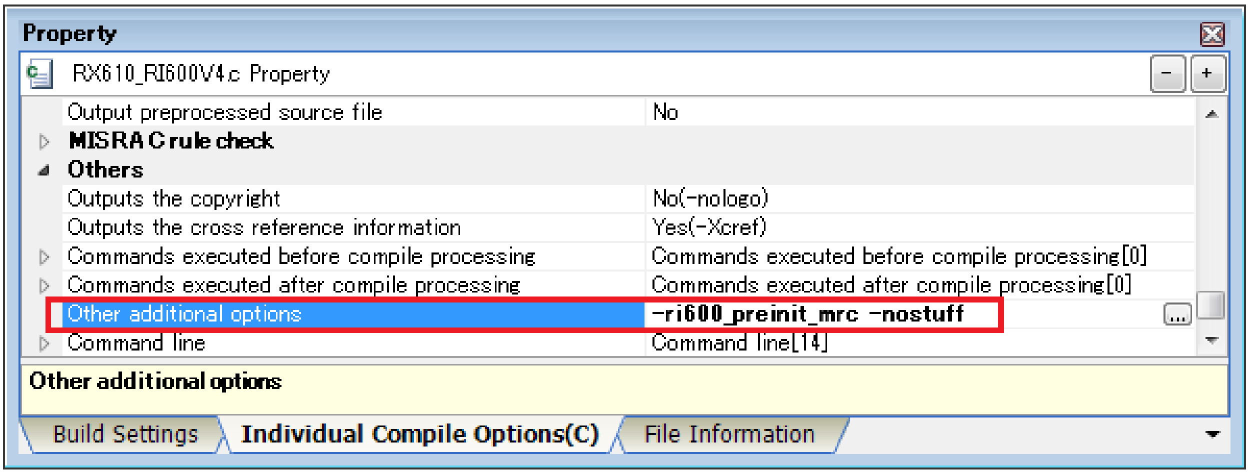
1) Set in the [Object] category
Like Figure 2-6, set “Yes” in [Allocates uninitialized variables to 4-byte boundary alignment sections], [Allocates initialized variables to 4-byte boundary alignment sections] and [Allocates const qualified variables to 4-byte boundary alignment sections].

Figure 2-6 [Object] category



2) Set in the [Others] category
Like Figure 2-7, add “-nostuff” to [Other additional options].

Figure 2-7 [Others] category



2.6.3 Kernel library

The kernel libraries are stored in the folders described in Table 2-1. Note, CubeSuite+ links the appropriate kernel library automatically, you need not consider the kernel libraries.

Table 2-1 Kernel libraries

	Folder	Compiler version corresponding to the library	Corresponding CPU core	File name	Description
1	<ri_root>\library\rxv1	V1.02.01 or later	RXv1 architecture	ri600lit.lib	For little endian
				ri600big.lib	For big endian
2	<ri_root>\library\rxv2	V2.01.00 or later	RXv1 architecture and RXv2 architecture	ri600lit.lib	For little endian
				ri600big.lib	For big endian

Note 1 <ri_root> indicates the installation folder of RI600V4.
The default folder is “C:\Program Files\Renesas Electronics\CubeSuite+\RI600V4”.

Note 2 The kernel described in item-2 of Table 2-1 is linked when compiler V2.01.00 or later is used. In the case of others, the kernel library described in item-1 of Table 2-1 is linked.

2.6.4 Arrangement of section

Arrangement section is defined by using "-start" linker option. In CubeSuite+, it is set in [Section] category of [Link Options] tab in [Property] panel for [CC-RX (Build Tool)].

1) RI600V4 sections
Table 2-2 shows RI600V4 sections.

Table 2-2 RI600V4 sections

Section name	Attribute	Boundary alignment	ROM/RAM	Description
PRI_KERNEL	CODE	1	ROM/RAM	RI600V4 programs
CRI_ROM	ROMDATA	4	ROM/RAM	RI600V4 constant
FIX_INTERRUPT_VECTOR	ROMDATA	4	ROM	Fixed vector table/Exception vector table Refer to "FIX_INTERRUPT_VECTOR section"
INTERRUPT_VECTOR	ROMDATA	4	ROM/RAM	Relocatable vector table (1KB)
BRI_RAM	DATA	4	RAM	RI600V4 variable section This section includes data queue area.
DRI_ROM	ROMDATA	4	ROM/RAM	RI600V4's initialized data. The size is 4 bytes.
RRI_RAM	DATA	4	RAM	
BRI_TRCBUF	DATA	4	RAM	This section is generated only when "Taking in trace chart by software trace mode" and "Kernel buffer" are selected in [Task Analyzer] tab. The size is specified in [Task Analyzer] tab.
BRI_HEAP	DATA	4	RAM	The section name assigned to message buffer area, fixed-sized memory pool area and variable-sized memory pool area can be specified in the system configuration file. When this is omitted, BRI_HEAP is applied as the section name.
SI	DATA	4	RAM	System stack
SURI_STACK	DATA	4	RAM	The section name assigned to the user stack for tasks can be specified in the system configuration file. When this is omitted, SURI_STACK is applied as the section name.

2) FIX_INTERRUPT_VECTOR section

The configurator cfg600 generates fixed vector table/exception vector table as FIX_INTERRUPT_VECTOR section according to the contents of definitions of “interrupt_fvector[]” in the system configuration file.

- At the time of RXv1 architecture use
  In the RXv1 architecture, fixed vector table is being fixed to address 0xFFFFFF80. It is necessary to arrange the FIX_INTERRUPT_VECTOR section at address 0xFFFFFF80.
  When the FIX_INTERRUPT_VECTOR section is not arranged to address 0xFFFFFF80, all “interrupt_fvector[]” in the system configuration file are ignored. And the system-down function when an exception (except Reset) assigned to fixed vector table is occurred does not operate normally. Please generate fixed vector table to address 0xFFFFFF80 by the user side.

- At the time of RXv2 architecture use
  In the RXv2 architecture, the name of fixed vector table is changed into exception vector table, and can set up the start address by EXTB register. The initial value of EXTB register at the time of reset is 0xFFFFFF80, it is same as fixed interrupt vector table in RXv1 architecture.
  Usually, please arrange the FIX_INTERRUPT_VECTOR section to address 0xFFFFFF80.
  When the FIX_INTERRUPT_VECTOR section is not arranged to address 0xFFFFFF80, “interrupt_fvector[31]” (reset vector) in the system configuration file is ignored. Please generate the reset vector (address 0xFFFFFFFC) by the user side. And initialize EXTB register to the start address of FIX_INTERRUPT_VECTOR section in Boot processing function (PowerON_Reset_PC( )).

3) Attention concerning address 0

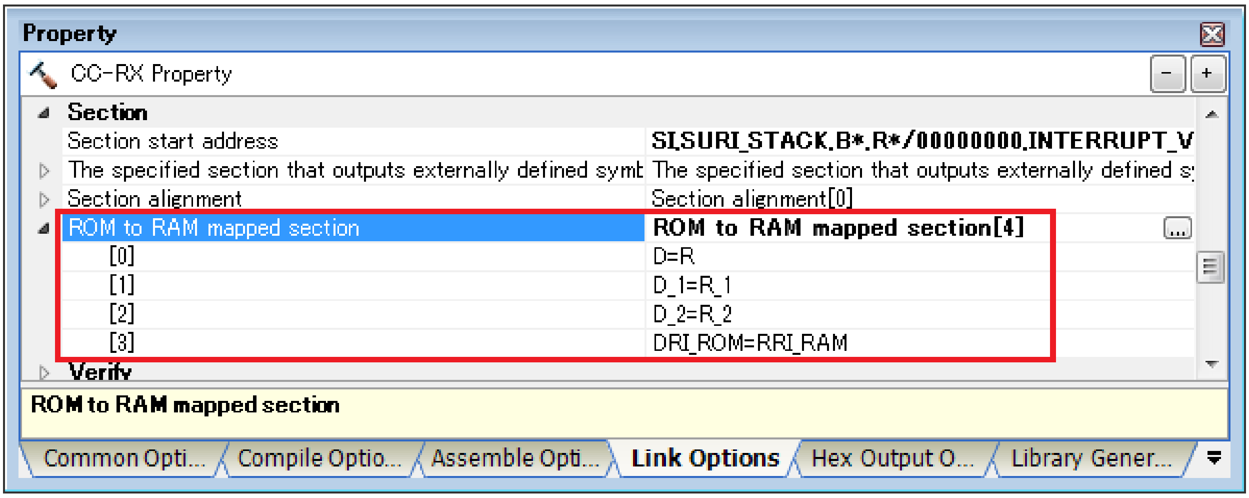
The following must not become address 0.

- Fixed-sized memory pool area
- Variable-sized memory pool area
- Message sent to a mailbox

2.6.5 Initialized data section

About sections described in DTBL of the Section information file (User-Own Coding Module), it is necessary to perform setting to map sections placed on ROM to sections placed on RAM by using “-rom” linker option. Set [Link Options] tab of [Property] panel for [CC-RX (Build Tool)] like Figure 2-8.

Figure 2-8 ROM to RAM mapped section



Note In sample projects provided by RI600V4, it is already set up that the “DRI_ROM” section of RI600V4 is mapped to “RRI_RAM” section.

2.6.6 Options for Realtime OS Task Analyzer

According to a setup of [Task Analyzer] tab, the build-options shown in Table 2-3 are set up automatically. Note, this automatic setting function is not being interlocked with corresponding property panel of a function. For this reason, don't change the contents set up automatically in corresponding property panel of a function.

Table 2-3 The options set up automatically for Realtime OS Task Analyzer

Trace mode	Assembler Options	Linker Options
Taking in trace chart by hardware trace mode	`-define=TRCMODE=1`	None

Taking in trace chart by software trace mode, Kernel buffer	`-define=TRCMODE=2`
	`-define=TRCBUFSZ=`*<Buffer size>*
	The following is also set up when “Stop the trace taking in” is chosen as “Operation after used up the buffers”.
	`-define=TRCBUFMODE=1`
		None

Taking in trace chart by software trace mode, Another buffer	`-define=TRCMODE=2`
	The following is also set up when “Stop the trace taking in” is chosen as “Operation after used up the buffers”.
	`-define=TRCBUFMODE=1`
		`-define=__RI_TRCBUF=`*<Buffer address>*
		`-define=__RI_TRCBUFSZ=`*<Buffer size>*

Taking in long-statistics by software trace mode	`-define=TRCMODE=3`	None
Not tracing	None	None

Note 1 The “TRCMODE” is used by following files.

- ritable.src: This file is generated by the configurator cfg600.
- trcSW_cmt.src: User-own coding sample module for “Taking in trace chart by software trace mode”
- trcLONG_cmt.src: User-own coding sample module for “Taking in long-statistics by software trace mode”

Note 2 The “TRCBUFSZ” and “TRCBUFMODE” are used by “ritable.src”.

CHAPTER 3 TASK MANAGEMENT FUNCTIONS

This chapter describes the task management functions performed by the RI600V4.

3.1 Outline

The task management functions provided by the RI600V4 include a function to reference task statuses such as priorities and detailed task information, in addition to a function to manipulate task statuses such as generation, activation and termination of tasks.

3.2 Tasks

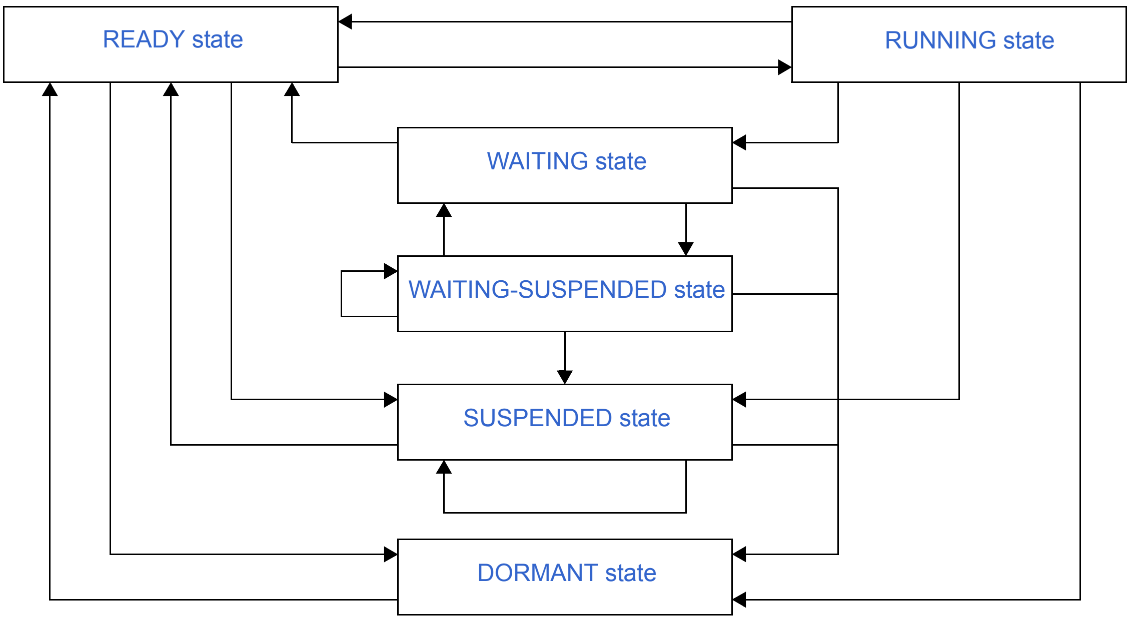
A task is processing program that is not executed unless it is explicitly manipulated via service calls provided by the RI600V4, unlike other processing programs (interrupt handler, cyclic handler and alarm handler), and is called from the scheduler.

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 RI600V4 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 RI600V4.
The RI600V4 classifies task states into the following six types.

Figure 3-1 Task State



1) 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 RI600V4, is not subject to RI600V4 scheduling.

2) 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.

3) 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.

4) 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 RI600V4, the WAITING state is classified into the following 12 types according to their required conditions and managed.

Table 3-1 WAITING State

WAITING State	Service Calls
Sleeping state	slp_tsk or tslp_tsk.
Delayed state	dly_tsk.
WAITING state for a semaphore resource	wai_sem or twai_sem.
WAITING state for an eventflag	wai_flg or twai_flg.
Sending WAITING state for a data queue	snd_dtq or tsnd_dtq.
Receiving WAITING state for a data queue	rcv_dtq or trcv_dtq.
Receiving WAITING state for a mailbox	rcv_mbx or trcv_mbx.
WAITING state for a mutex	loc_mtx or tloc_mtx.
Sending WAITING state for a message buffer	snd_mbf or tsnd_mbf
Receiving WAITING state for a message buffer	rcv_mbf or trcv_mbf
WAITING state for a fixed-sized memory block	get_mpf or tget_mpf.
WAITING state for a variable-sized memory block	get_mpl or tget_mpl.

5) 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.

6) 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 RI600V4, 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 RI600V4, the following two types of priorities are used for management purposes.

- Current priority
  The RI600V4 performs the following processing according to current priority.
  - Task scheduling (Refer to "14.4 Task Scheduling Method")
  - Queuing tasks to a wait queue in the order of priority

  Note The current priority immediately after it moves from the DORMANT state to the READY state is specified at creating the task.

- Base priority
  Unless mutex is used, the base priority is the same as the current priority. When using mutex, refer to "6.2.2 Current priority and base priority".

Note 1 In the RI600V4, a task having a smaller priority number is given a higher priority.

Note 2 The priority range that can be specified in a system can be defined by Maximum task priority (priority) in System Information (system)) when creating a system configuration file.

3.2.3 Basic form of tasks

The following shows the basic form of tasks.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

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

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

Note 1 The following information is passed to *exinf*.

How to activate	*exinf*
ON is specified for TA_ACT attribute (initial_start) in Task Information (task[])	Extended information (exinf) defined in Task Information (task[])
act_tsk or iact_tsk	
sta_tsk or ista_tsk	Start code (*stacd*) specified by sta_tsk or ista_tsk

Note 2 When the return instruction is issued in a task, the same processing as ext_tsk is performed.

Note 3 For details about the extended information, refer to "3.4 Activate Task".

Note 4 The prototype for tasks are declared in the kernel_id.h which is generated by the cfg600.

3.2.4 Internal processing of task

In the RI600V4, original dispatch processing (task scheduling) is executed during task switching.
Therefore, note the following points when coding tasks.

- Stack
  Tasks use user stacks that are defined in Task Information (task[]).

- Service call
  Tasks can issue service calls whose “Useful range” is “Task”.

- PSW register when processing is started

Table 3-2 PSW Register When Task Processing is Started

Bit	Value	Note
I	1	All interrupts are acceptable.
IPL	0	
PM	1	User mode
U	1	User stack
C, Z, S, O	Undefined	
Others	0	

- FPSW register when processing is started
  When setting of Task context register (context) in System Information (system) includes “FPSW”, the FPSW when processing is started is shown in Table 3-3. The FPSW when processing is undefined in other cases.

Table 3-3 FPSW Register When Task Processing is Started

Compiler options		Value
-round	-denormalize	
nearest (default)	off (default)	0x00000100 (Only DN bit is 1.)
	on	0
zero	off (default)	0x00000101 (Only DN bit and RM bit are 1.)
	on	1 (Only RM bit is 1.)

3.2.5 Processor mode of task

The processor mode at the time of task execution is always user mode. It is impossible to execute a task in the supervisor mode.
Processing to execute in the supervisor mode should be implemented as an interrupt handler for INT instruction.
For example, the WAIT instruction, that changes the CPU to the power saving mode, is privilege instruction. The WAIT instruction should execute in the supervisor mode.
Note, INT #1 to #8 are reserved by the RI600V4, application cannot use INT #1 to #8.

3.3 Create Task

In the RI600V4, the method of creating a task is limited to “static creation”.
Tasks therefore cannot be created dynamically using a method such as issuing a service call from a processing program.
Static task creation means defining of tasks using static API “task[]” in the system configuration file.
For details about the static API “task[]”, refer to “19.7 Task Information (task[])”.

3.4 Activate Task

The RI600V4 provides two types of interfaces for task activation: queuing an activation request queuing and not queuing an activation request.

3.4.1 Activate task with queuing

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 the 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 RI600V4.
  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 1 to the activation request counter).
  The following describes an example for coding these service calls.

```
#include    "kernel.h"          /*Standard header file definition*/
#include    "kernel_id.h"       /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      tskid = 8;          /*Declares and initializes variable*/

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

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

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

Note 1 The activation request counter managed by the RI600V4 is configured in 8-bit widths. If the number of activation requests exceeds the maximum count value 255 as a result of issuing this service call, the counter manipulation processing is therefore not performed but “E_QOVR” is returned.

Note 2 Extended information specified in Task Information (task[]) is passed to the task activated by issuing these service calls.

3.4.2 Activate task without queuing

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 the 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 RI600V4.
  This service call does not perform queuing of activation requests. If the target task is in a state other than the DORMANT state, the status manipulation processing for the target task is therefore not performed but “E_OBJ” is returned.
  Specify for parameter *stacd* the extended information transferred to the target task.
  The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      tskid = 8;          /*Declares and initializes variable*/
    VP_INT  stacd = 123;        /*Declares and initializes variable*/

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

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

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

3.5 Cancel Task Activation Requests

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

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

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER_UINT ercd;                   /*Declares variable*/
    ID      tskid = 8;              /*Declares and initializes variable*/

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

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

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

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

Note This service call does not perform status manipulation processing but performs the setting of activation request counter. Therefore, the task does not move from a state such as the READY state to the DORMANT state.

3.6 Terminate Task

3.6.1 Terminate invoking task

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

- ext_tsk
  This service call moves the 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 RI600V4 scheduling subject.
  If an activation request has been queued to the invoking task (the activation request counter > 0) when this service call is issued, this service call moves the task from the RUNNING state to the DORMANT state, decrements the wake-up request counter (by subtracting 1 from the activation 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"           /*Header file generated by cfg600*/

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

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

Note 1 When the invoking task has locked mutexes, the locked state are released at the same time (processing equivalent to unl_mtx).

Note 2 When the return instruction is issued in a task, the same processing as ext_tsk is performed.

3.6.2 Terminate Another task

- ter_tsk
  This service call forcibly moves the task specified by parameter *tskid* to the DORMANT state.
  As a result, the target task is excluded from the RI600V4 scheduling subject.
  If an activation request has been queued to the target task (the activation request counter > 0) when this service call is issued, this service call moves the task to the DORMANT state, decrements the wake-up request counter (by subtracting 1 from the activation 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"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      tskid = 8;              /*Declares and initializes variable*/

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

    ter_tsk (tskid);                /*Terminate task*/

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

Note When the target task has locked mutexes, the locked state are released at the same time (processing equivalent to unl_mtx).

3.7 Change Task Priority

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

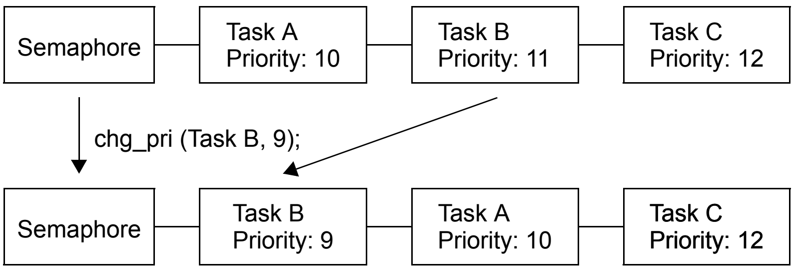
- chg_pri, ichg_pri

  This service call changes the base priority of the task specified by parameter *tskid* to a value specified by parameter *tskpri*.
  The changed base priority is effective until the task terminates or this servie call is issued. When next the task is activated, the base priority is the initial priority which is specified at the task creation.
  This service call also changes the current priority of the target task to a value specified by parameter *tskpri*. However, the current priority is not changed when the target task has locked mutexes.
  If the target task has locked mutexes or is waiting for mutex to be locked and if *tskpri* is higher than the ceiling priority of either of the mutexes, this service call returns “E_ILUSE”.
  When the current priority is changed, the following state variations are generated.

  1) When the target task is in the RUNNING or READY state.
  This service call re-queues the task at the end of the ready queue corresponding to the priority specified by parameter *tskpri*.

  2) When the target task is queued to a wait queue of the object with TA_TPRI or TA_CEILING attribute.
  This service call re-queues the task to the wait queue corresponding to the priority specified by parameter *tskpri*. When two or more tasks of same current priority as this service call re-queues the target task at the end among their tasks.

  Example When three tasks (task A: priority level 10, task B: priority level 11, task C: priority level 12) are queued to the semaphore wait queue in the order of priority, and the priority level of task B is changed from 11 to 9, the wait order will be changed as follows.



  The following describes an example for coding these service calls.

```
#include    "kernel.h"               /*Standard header file definition*/
#include    "kernel_id.h"            /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      tskid = 8;               /*Declares and initializes variable*/
    PRI     tskpri = 9;              /*Declares and initializes variable*/

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

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

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

  Note For current priority and base priority, refer to “6.2.2 Current priority and base priority”.

3.8 Reference Task Priority

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

- get_pri, iget_pri
  Stores current priority of the task specified by parameter *tskid* in the area specified by parameter *p_tskpri*.
  The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      tskid = 8;              /*Declares and initializes variable*/
    PRI     p_tskpri;               /*Declares variable*/

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

    get_pri (tskid, &p_tskpri);     /*Reference task priority*/

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

Note For current priority and base priority, refer to "6.2.2 Current priority and base priority".

3.9 Reference Task State

3.9.1 Reference task state

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

- ref_tsk, iref_tsk
  Stores task state packet (current state, current priority, etc.) of the task specified by parameter *tskid* in the area specified by parameter *pk_rtsk*.
  The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      tskid = 8;              /*Declares and initializes variable*/
    T_RTSK  pk_rtsk;                /*Declares data structure*/
    STAT    tskstat;                /*Declares variable*/
    PRI     tskpri;                 /*Declares variable*/
    PRI     tskbpri;                /*Declares variable*/
    STAT    tskwait;                /*Declares variable*/
    ID      wobjid;                 /*Declares variable*/
    TMO     lefttmo;                /*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 current state*/
    tskpri = pk_rtsk.tskpri;        /*Reference current priority*/
    tskbpri = pk_rtsk.tskbpri;      /*Reference base priority*/
    tskwait = pk_rtsk.tskwait;      /*Reference reason for waiting*/
    wobjid = pk_rtsk.wobjid;        /*Reference object ID number for which the */
                                    /*task is waiting*/
    lefttmo = pk_rtsk.lefttmo;      /*Reference remaining time until time-out*/
    actcnt = pk_rtsk.actcnt;        /*Reference activation request count*/
    wupcnt = pk_rtsk.wupcnt;        /*Reference wake-up request count*/
    suscnt = pk_rtsk.suscnt;        /*Reference suspension count*/

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

Note For details about the task state packet, refer to "[Task state packet: T_RTSK]".

3.9.2 Reference task state (simplified version)

A task status (simplified version) is referenced by issuing the following service call from the processing program.

- ref_tst, iref_tst
  Stores task state packet (current state, reason for waiting) of the task specified by parameter *tskid* in the area specified by parameter *pk_rtst*.
  Used for referencing only the current state and reason for wait among task information.
  Response becomes faster than using ref_tsk or iref_tsk because only a few information items are acquired.
  The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      tskid = 8;              /*Declares and initializes variable*/
    T_RTST  pk_rtst;                /*Declares data structure*/
    STAT    tskstat;                /*Declares variable*/
    STAT    tskwait;                /*Declares variable*/

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

    ref_tst (tskid, &pk_rtst);      /*Reference task state (simplified version)*/

    tskstat = pk_rtst.tskstat;      /*Reference current state*/
    tskwait = pk_rtst.tskwait;      /*Reference reason for waiting*/

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

Note For details about the task state packet (simplified version), refer to " [Task state packet (simplified version): T_RTST]".

CHAPTER 4 TASK DEPENDENT SYNCHRONIZATION FUNCTIONS

This chapter describes the task dependent synchronization functions performed by the RI600V4.

4.1 Outline

The RI600V4 provides several task-dependent synchronization functions.

4.2 Put Task to Sleep

4.2.1 Waiting forever

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

- slp_tsk
  This service call moves the invoking task from the RUNNING state to the WAITING state (sleeping state).
  If a wake-up request has been queued to the target task (the wake-up request counter > 0) when this service call is issued, this service call does not move the state but decrements the wake-up request counter (by subtracting 1 from the wake-up request counter).
  The sleeping state is cancelled in the following cases.

Sleeping State Cancel Operation	Return Value
A wake-up request was issued as a result of issuing wup_tsk.	E_OK
A wake-up 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"          /*Header file generated by cfg600*/

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

    /* ......... */
    ercd = slp_tsk ();             /*Put task to sleep*/
    if (ercd == E_OK) {
        /* ......... */            /*Normal termination processing*/
    } else if (ercd == E_RLWAI) {
        /* ......... */            /*Forced termination processing*/
    }
    /* ......... */
}
```

4.2.2 With time-out

A task is moved to the sleeping state (with time-out) by issuing the following service call from the processing program.

- tslp_tsk
  This service call moves the invoking task from the RUNNING state to the WAITING state with time-out(sleeping state).
  As a result, the invoking task is unlinked from the ready queue and excluded from the RI600V4 scheduling subject.
  If a wake-up request has been queued to the target task (the wake-up request counter > 0) when this service call is issued, this service call does not move the state but decrements the wake-up request counter (by subtracting 1 from the wake-up request counter).
  The sleeping state is cancelled in the following cases.

Sleeping State Cancel Operation	Return Value
A wake-up request was issued as a result of issuing wup_tsk.	E_OK
A wake-up 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 time specified by *tmout* has elapsed.	E_TMOUT

The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

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

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

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

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

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

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

4.3 Wake-up 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 wake-up request counter (by added 1 to the wake-up request counter).
  The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      tskid = 8;              /*Declares and initializes variable*/

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

    wup_tsk (tskid);                /*Wake-up task*/

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

Note The wake-up request counter managed by the RI600V4 is configured in 8-bit widths. If the number of wake-up requests exceeds the maximum count value 255 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 Wake-up Requests

A wake-up request is cancelled by issuing the following service call from the processing program.

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

```
#include    "kernel.h"                /*Standard header file definition*/
#include    "kernel_id.h"             /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER_UINT ercd;                     /*Declares variable*/
    ID      tskid = 8;                /*Declares and initializes variable*/

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

    ercd = can_wup (tskid);           /*Cancel task wake-up requests*/

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

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

4.5 Forcibly 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 these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      tskid = 8;              /*Declares and initializes variable*/

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

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

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

Note 1 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.

Note 2 The SUSPENDED state is not cancelled by these service calls.

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 move the target task specified by parameter *tskid* 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, these service calls return E_QOVR.
  The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      tskid = 8;              /*Declares and initializes variable*/

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

    sus_tsk (tskid);                /*Suspend task*/

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

4.7 Resume Suspended Task

4.7.1 Resume suspended task

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

- rsm_tsk, irsm_tsk
  These service calls move the target task specified by parameter *tskid* 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 these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      tskid = 8;              /*Declares and initializes variable*/

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

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

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

Note 1 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 returned.

Note 2 The RI600V4 does not support queuing of suspend request. The behavior of the frsm_tsk and ifrsm_tsk, that can release from the SUSPENDED state even if suspend request has been queued, are same as rsm_tsk and irsm_tsk.

4.7.2 Forcibly resume suspended task

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

- frsm_tsk, ifrsm_tsk
  These service calls move the target task specified by parameter *tskid* 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 these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      tskid = 8;              /*Declares and initializes variable*/

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

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

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

Note 1 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.

Note 2 The RI600V4 does not support queuing of suspend request. Therefore, the behavior of these service calls are same as rsm_tsk and irsm_tsk.

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 RI600V4 scheduling subject.
  The delayed state is cancelled in the following cases.

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"           /*Header file generated by cfg600*/

void 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*/
    }

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

Note When 0 is specified as *dlytim*, the delay time is up to next base clock interrupt generation.

4.9 Differences Between Sleep with Time-out and Delay

There are differences between “Sleep with time-out (4.2.2 With time-out)” and “Delay (4.8 Delay Task)” as shown in Table 4-1.

Table 4-1 Differences Between “Sleep with time-out” and “Delay”

	Sleep with time-out	Delay
Service call that causes status change	tslp_tsk	dly_tsk
Return value when time has elapsed	E_TMOUT	E_OK
Operation when wup_tsk or iwup_tsk is issued	Wake-up	Queues the wake-up request (time elapse wait is not cancelled).

CHAPTER 5 SYNCHRONIZATION AND COMMUNICATION FUNCTIONS

This chapter describes the synchronization and communication functions performed by the RI600V4.

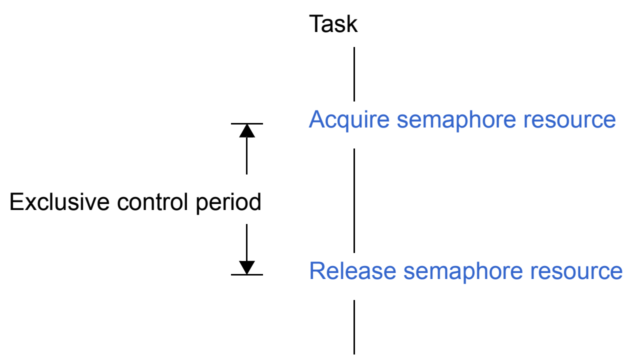
5.1 Outline

The synchronization and communication functions of the RI600V4 consist of Semaphores, Eventflags, Data Queues, and Mailboxes that are provided as means for realizing exclusive control, queuing, and communication among tasks.

5.2 Semaphores

In the RI600V4, 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 RI600V4, the method of creating a semaphore is limited to “static creation”.
Semaphores therefore cannot be created dynamically using a method such as issuing a service call from a processing program.
Static semaphore creation means defining of semaphores using static API “semaphore[]” in the system configuration file.
For details about the static API “semaphore[]”, refer to “19.8 Semaphore Information (semaphore[])”.

5.2.2 Acquire semaphore resource

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

- wai_sem (Wait)
- pol_sem, ipol_sem (Polling)
- twai_sem (Wait with time-out)

- wai_sem (Wait)
  This service call acquires a resource from the semaphore specified by parameter *semid* (subtracts 1 from the semaphore counter).
  When no resources are acquired from the target semaphore when this service call is issued (no available resources exist), this service call does not acquire resources but queues the invoking task to the target semaphore wait queue and moves it from the RUNNING state to the WAITING state (resource acquisition wait state).
  The WAITING state for a semaphore resource is cancelled in the following cases.

WAITING State for a Semaphore Resource Cancel Operation	Return Value
The resource was released to the target semaphore as a result of issuing sig_sem.	E_OK
The resource was released 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"           /*Header file generated by cfg600*/

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

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

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

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

        sig_sem ( semid );      /*Release semaphore resource*/
    } else if (ercd == E_RLWAI) {
        /* ......... */         /*Forced termination processing*/
    }

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

Note Invoking tasks are queued to the target semaphore wait queue in the order defined during configuration (FIFO order or current priority order).

- pol_sem, ipol_sem (Polling)
  These service calls acquire a resource from the semaphore specified by parameter *semid* (subtracts 1 from the semaphore counter).
  If a resource could not be acquired from the target semaphore (semaphore counter is set to 0) when these service calls are issued, the counter manipulation processing is not performed but “E_TMOUT” is returned.
  The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

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

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

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

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

        sig_sem ( semid );          /*Release semaphore resource*/
    } else if (ercd == E_TMOUT) {
        /* ......... */             /*Polling failure processing*/
    }

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

- twai_sem (Wait with time-out)
  This service call acquires a resource from the semaphore specified by parameter *semid* (subtracts 1 from the semaphore counter).
  If no resources are acquired from the target semaphore when service call is issued this (no available resources exist), this service call does not acquire resources but queues the invoking task to the target semaphore wait queue and moves it from the RUNNING state to the WAITING state with time-out (resource acquisition wait state).
  The WAITING state for a semaphore resource is cancelled in the following cases.

WAITING State for a Semaphore Resource Cancel Operation	Return Value
The resource was released to the target semaphore as a result of issuing sig_sem.	E_OK
The resource was released 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 time specified by *tmout* has elapsed.	E_TMOUT

The following describes an example for coding this service call.

```
#include     "kernel.h"                  /*Standard header file definition*/
#include     "kernel_id.h"               /*Header file generated by cfg600*/

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

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

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

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

        sig_sem ( semid );               /*Release semaphore resource*/
    } else if (ercd == E_RLWAI) {
        /* ......... */                  /*Forced termination processing*/
    } else if (ercd == E_TMOUT) {
        /* ......... */                  /*Time-out processing*/
    }

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

Note 1 Invoking tasks are queued to the target semaphore wait queue in the order defined during configuration (FIFO order or current priority order).

Note 2 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.

5.2.3 Release semaphore resource

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

- sig_sem, isig_sem
  These service calls releases the resource to the semaphore specified by parameter *semid* (adds 1 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 these service calls.

```
#include    "kernel.h"          /*Standard header file definition*/
#include    "kernel_id.h"       /*Header file generated by cfg600*/

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

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

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

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

        sig_sem ( semid );      /*Release semaphore resource*/
    } else if (ercd == E_RLWAI) {
        /* ......... */         /*Forced termination processing*/
    }

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

Note With the RI600V4, the maximum possible number of semaphore resources (maximum resource count) is defined during configuration. If the number of resources exceeds the specified maximum resource count, this service call therefore does not release the acquired resources (addition to the semaphore counter value) but returns E_QOVR.

5.2.4 Reference semaphore state

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

- ref_sem, iref_sem
  Stores semaphore state packet (ID number of the task at the head of the wait queue, current resource count, etc.) of the semaphore specified by parameter *semid* in the area specified by parameter *pk_rsem*.
  The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      semid = 1;              /*Declares and initializes variable*/
    T_RSEM  pk_rsem;                /*Declares variable*/
    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 “[Semaphore state packet: T_RSEM]”.

5.3 Eventflags

The RI600V4 provides 32-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 RI600V4, the method of creating an eventflag is limited to "static creation".
Eventflags therefore cannot be created dynamically using a method such as issuing a service call from a processing program.
Static event flag creation means defining of event flags using static API "flag[]" in the system configuration file.
For details about the static API "flag[]", refer to "19.9 Eventflag Information (flag[])".

5.3.2 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.
  After that, these service calls evaluate whether the wait condition of the tasks in the wait queue is satisfied. This evaluation is done in order of the wait queue. If the wait condition is satisfied, 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. At this time, the bit pattern of the target event flag is cleared to 0 and this service call finishes processing if the TA_CLR attribute is specified for the target eventflag.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      flgid = 1;              /*Declares and initializes variable*/
    FLGPTN  setptn = 0x00000001UL;  /*Declares and initializes variable*/

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

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

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

5.3.3 Clear eventflag

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

- clr_flg, iclr_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 these service calls.

```
#include    "kernel.h"               /*Standard header file definition*/
#include    "kernel_id.h"            /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      flgid = 1;               /*Declares and initializes variable*/
    FLGPTN  clrptn = 0xFFFFFFFEUL;   /*Declares and initializes variable*/

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

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

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

5.3.4 Check bit pattern

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

- wai_flg (Wait)
- pol_flg, ipol_flg (Polling)
- twai_flg (Wait with time-out)

- wai_flg (Wait)
  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.

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"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      flgid = 1;              /*Declares and initializes variable*/
    FLGPTN  waiptn = 14;            /*Declares and initializes variable*/
    MODE    wfmode = TWF_ANDW;      /*Declares and initializes variable*/
    FLGPTN  p_flgptn;               /*Declares variable*/

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

                                    /*Wait for eventflag*/
    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 With the RI600V4, whether to enable queuing of multiple tasks to the event flag wait queue is defined during configuration. If this service call is issued for the event flag (TA_WSGL attribute) to which a wait task is queued, therefore, "E_ILUSE" is returned regardless of whether the required condition is immediately satisfied.

TA_WSGL:	Only one task is allowed to be in the WAITING state for the eventflag.
TA_WMUL:	Multiple tasks are allowed to be in the WAITING state for the eventflag.

Note 2 Invoking tasks are queued to the target event flag (TA_WMUL attribute) wait queue in the order defined during configuration (FIFO order or current priority order).
However, when the TA_CLR attribute is not specified, the wait queue is managed in the FIFO order even if the priority order is specified. This behavior falls outside μITRON4.0 specification.

Note 3 The RI600V4 performs bit pattern clear processing (0 setting) when the required condition of the target eventflag (TA_CLR attribute) is satisfied.

- pol_flg, ipol_flg (Polling)
  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 these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      flgid = 1;              /*Declares and initializes variable*/
    FLGPTN  waiptn = 14;            /*Declares and initializes variable*/
    MODE    wfmode = TWF_ANDW;      /*Declares and initializes variable*/
    FLGPTN  p_flgptn;               /*Declares variable*/

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

                                    /*Wait for eventflag*/
    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 With the RI600V4, whether to enable queuing of multiple tasks to the event flag wait queue is defined during configuration. If this service call is issued for the event flag (TA_WSGL attribute) to which a wait task is queued, therefore, “E_ILUSE” is returned regardless of whether the required condition is immediately satisfied.

TA_WSGL:	Only one task is allowed to be in the WAITING state for the eventflag.
TA_WMUL:	Multiple tasks are allowed to be in the WAITING state for the eventflag.

Note 2 The RI600V4 performs bit pattern clear processing (0 setting) when the required condition of the target eventflag (TA_CLR attribute) is satisfied.

- twai_flg (Wait with time-out)
  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.

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 time specified by *tmout* has elapsed.	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"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      flgid = 1;              /*Declares and initializes variable*/
    FLGPTN  waiptn = 14;            /*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*/
    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) {
        /* ......... */             /*Time-out processing*/
    }

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

Note 1 With the RI600V4, whether to enable queuing of multiple tasks to the event flag wait queue is defined during configuration. If this service call is issued for the event flag (TA_WSGL attribute) to which a wait task is queued, therefore, "E_ILUSE" is returned regardless of whether the required condition is immediately satisfied.

TA_WSGL: Only one task is allowed to be in the WAITING state for the eventflag.
TA_WMUL: Multiple tasks are allowed to be in the WAITING state for the eventflag.

Note 2 Invoking tasks are queued to the target event flag (TA_WMUL attribute) wait queue in the order defined during configuration (FIFO order or current priority order).
However, when the TA_CLR attribute is not specified, the wait queue is managed in the FIFO order even if the priority order is specified. This behavior falls outside μITRON4.0 specification.

Note 3 The RI600V4 performs bit pattern clear processing (0 setting) when the required condition of the target eventflag (TA_CLR attribute) is satisfied.

Note 4 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.

5.3.5 Reference eventflag state

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

- ref_flg, iref_flg
  Stores eventflag state packet (ID number of the task at the head of the wait queue, current bit pattern, etc.) of the eventflag specified by parameter *flgid* in the area specified by parameter *pk_rflg*.
  The following describes an example for coding these service calls.

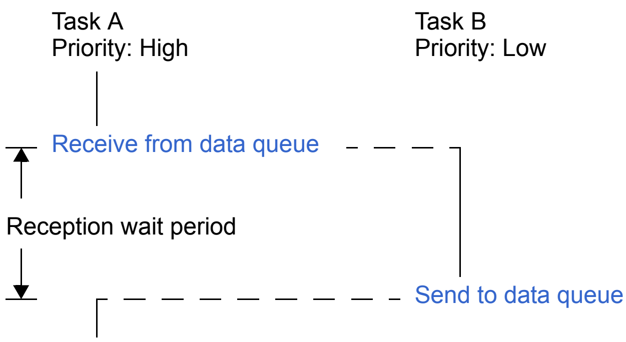
```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      flgid = 1;              /*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 "[Eventflag state packet: T_RFLG]".

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 RI600V4 therefore provides the data queues 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 RI600V4, the method of creating data 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 “dataqueue[]” in the system configuration file.
For details about the static API “dataqueue[]”, refer to “19.10 Data Queue Information (dataqueue[])”.

5.4.2 Send to data queue

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

- snd_dtq (Wait)
- psnd_dtq, ipsnd_dtq (Polling)
- tsnd_dtq (Wait with time-out)

- snd_dtq (Wait)
  This service call processes as follows according to the situation of the data queue specified by the parameter *dtqid*.

  - There is a task in the reception wait queue.
    This service call transfers the data specified by parameter *data* to the task in the top of the reception wait queue. 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.

  - There is no task neither in the reception wait queue and transmission wait queue and there is available space in the data queue.
    This service call stores the data specified by parameter *data* to the data queue.

  - There is no task neither in the reception wait queue and transmission wait queue and there is no available space in the data queue, or there is a task in the transmission wait queue.
    This service call 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.

Sending WAITING State for a Data Queue Cancel Operation	Return Value
Available space was secured in the data queue area as a result of issuing rcv_dtq.	E_OK
Available space was secured in the data queue area as a result of issuing prcv_dtq.	E_OK
Available space was secured in the data queue area as a result of issuing iprcv_dtq.	E_OK
Available space was secured in the data queue area 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
The data queue is reset as a result of issuing vrst_dtq.	EV_RST

The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

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*/

    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 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 or current priority order).

- psnd_dtq, ipsnd_dtq (Polling)
  These service calls process as follows according to the situation of the data queue specified by the parameter *dtqid*.
    - There is a task in the reception wait queue.
      These service calls transfer the data specified by parameter *data* to the task in the top of the reception wait queue. 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.
    - There is no task neither in the reception wait queue and transmission wait queue and there is available space in the data queue.
      These service calls store the data specified by parameter *data* to the data queue.
    - There is no task neither in the reception wait queue and transmission wait queue and there is no available space in the data queue, or there is a task in the transmission wait queue.
      These service calls return "E_TMOUT".

The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

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

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

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

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

- tsnd_dtq (Wait with time-out)
  This service call processes as follows according to the situation of the data queue specified by the parameter *dtqid*.

  - There is a task in the reception wait queue.
    This service call transfers the data specified by parameter *data* to the task in the top of the reception wait queue. 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.

  - There is no task neither in the reception wait queue and transmission wait queue and there is available space in the data queue.
    This service call stores the data specified by parameter *data* to the data queue.

  - There is no task neither in the reception wait queue and transmission wait queue and there is no available space in the data queue, or there is a task in the transmission wait queue.
    This service call 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.

Sending WAITING State for a Data Queue Cancel Operation	Return Value
Available space was secured in the data queue area as a result of issuing rcv_dtq.	E_OK
Available space was secured in the data queue area as a result of issuing prcv_dtq.	E_OK
Available space was secured in the data queue area as a result of issuing iprcv_dtq.	E_OK
Available space was secured in the data queue area 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
The data queue is reset as a result of issuing vrst_dtq.	EV_RST
The time specified by *tmout* has elapsed.	E_TMOUT

The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

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*/
    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) {
        /* ......... */             /*Time-out processing*/
    }

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

Note 1 Data is written to the data queue area 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 or current priority 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 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
  This service call processes as follows according to the situation of the data queue specified by the parameter *dtqid*.
  - There is a task in the reception wait queue.
    This service call transfers the data specified by parameter *data* to the task in the top of the reception wait queue. 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.
  - There is no task neither in the reception wait queue and transmission wait queue.
    This service call stores the data specified by parameter *data* to the data queue.
    If there is no available space in the data queue, this service call deletes the oldest data in the data queue before storing the data specified by *data* to the data queue.

The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

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*/

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

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

5.4.4 Receive from data queue

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

- rcv_dtq (Wait)
- prcv_dtq, iprcv_dtq (Polling)
- trcv_dtq (Wait with time-out)

- rcv_dtq (Wait)
  This service call processes as follows according to the situation of the data queue specified by the parameter *dtqid*.

  - There is a data in the data queue.
    This service call takes out the oldest data from the data queue and stores the data to the area specified by *p_data*.
    When there is a task in the transmission wait queue, this service call stores the data sent by the task in the top of the transmission wait queue and moves it from the WAITING state (data transmission wait state) to the READY state.

  - There is no data in the data queue and there is a task in the transmission wait queue.
    This service call stores the data specified by the task in the top of the transmission wait queue to the area specified by *p_data*. As a result, the task is unlinked from the transmission wait queue and moves from the WAITING state (data transmission wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
    Note, this situation is caused only when the capacity of the data queue is 0.

  - There is no data in the data queue and there is no task in the transmission wait queue.
    This service call 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.

Receiving WAITING State for a Data Queue Cancel Operation	Return Value
Data was sent to the data queue area as a result of issuing snd_dtq.	E_OK
Data was sent to the data queue area as a result of issuing psnd_dtq.	E_OK
Data was sent to the data queue area as a result of issuing ipsnd_dtq.	E_OK
Data was sent to the data queue area as a result of issuing tsnd_dtq.	E_OK
Data was sent to the data queue area as a result of issuing fsnd_dtq.	E_OK
Data was sent to the data queue area 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"           /*Header file generated by cfg600*/

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*/
    ercd = rcv_dtq (dtqid, &p_data);

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

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

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

- prcv_dtq, iprcv_dtq (Polling)
  These service calls process as follows according to the situation of the data queue specified by the parameter *dtqid*.

  - There is a data in the data queue.
    This service call takes out the oldest data from the data queue and stores the data to the area specified by *p_data*.
    When there is a task in the transmission wait queue, this service call stores the data sent by the task in the top of the transmission wait queue and moves it from the WAITING state (data transmission wait state) to the READY state.

  - There is no data in the data queue and there is a task in the transmission wait queue.
    These service calls store the data specified by the task in the top of the transmission wait queue to the area specified by *p_data*. As a result, the task is unlinked from the transmission wait queue and moves from the WAITING state (data transmission wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
    Note, this situation is caused only when the capacity of the data queue is 0.

  - There is no data in the data queue and there is no task in the transmission wait queue.
    These service calls return "E_TMOUT".

The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

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*/
    ercd = prcv_dtq (dtqid, &p_data);

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

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

- trcv_dtq (Wait with time-out)
  This service call processes as follows according to the situation of the data queue specified by the parameter *dtqid*.

  - There is a data in the data queue.
    This service call takes out the oldest data from the data queue and stores the data to the area specified by *p_data*.
    When there is a task in the transmission wait queue, this service call stores the data sent by the task in the top of the transmission wait queue and moves it from the WAITING state (data transmission wait state) to the READY state.

  - There is no data in the data queue and there is a task in the transmission wait queue.
    This service call stores the data specified by the task in the top of the transmission wait queue to the area specified by *p_data*. As a result, the task is unlinked from the transmission wait queue and moves from the WAITING state (data transmission wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
    Note, this situation is caused only when the capacity of the data queue is 0.

  - There is no data in the data queue and there is no task in the transmission wait queue.
    This service call 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 (data reception wait state).
    The receiving WAITING state for a data queue is cancelled in the following cases.

Receiving WAITING State for a Data Queue Cancel Operation	Return Value
Data was sent to the data queue area as a result of issuing snd_dtq.	E_OK
Data was sent to the data queue area as a result of issuing psnd_dtq.	E_OK
Data was sent to the data queue area as a result of issuing ipsnd_dtq.	E_OK
Data was sent to the data queue area as a result of issuing tsnd_dtq.	E_OK
Data was sent to the data queue area as a result of issuing fsnd_dtq.	E_OK
Data was sent to the data queue area 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 time specified by *tmout* has elapsed.	E_TMOUT

The following describes an example for coding this service call.

```
#include    "kernel.h"             /*Standard header file definition*/
#include    "kernel_id.h"          /*Header file generated by cfg600*/

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*/
    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) {
            /* ......... */        /*Time-out 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 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, iref_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 these service calls.

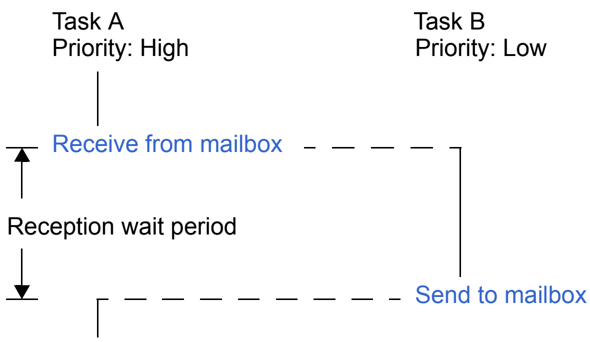
```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

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*/

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

    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*/

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

Note For details about the data queue state packet, refer to "[Data queue state packet: T_RDTQ]".

5.5 Mailboxes

Multitask processing requires the inter-task communication function (message transfer function) that reports the processing result of a task to another task. The RI600V4 therefore provides the mailbox for transferring the start address of a message written in the shared memory area.
The following shows a processing flow when using a mailbox.

Figure 5-4 Processing Flow (Mailbox)



5.5.1 Messages

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 RI600V4, 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.

- Message area
  In the case of the RI600V4, it is recommended to use the memory area secured by issuing get_mpf and get_mpl for messages.

- Basic form of messages
  In the RI600V4, 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 message must be started from the T_MSG structure. This area is used by the kernel. The use message should be arranged following the T_MSG structure.
    The length of the message is prescribed among the processing programs that exchange data using the mailbox.
    The following shows the basic form of coding TA_MFIFO attribute messages.

    [Message packet for TA_MFIFO attribute]

```
/* T_MSG structure, which is defined in the kernel.h*/
typedef struct {
    VP      msghead;       /*RI600V4 management area*/
} T_MSG;

/* Message structure defined by user*/
typedef struct {
    T_MSG   t_msg;         /*T_MSG structure*/
    B       data[8];       /*User message*/
} USER_MSG;
```

  - When using a mailbox with the TA_MPRI attribute
    The message must be started from the T_MSG_PRI structure. The T_MSG_PRI.msgque is used by the kernel.
    The message priority should be set to T_MSG_PRI.msgpri.
    The length of the message is prescribed among the processing programs that exchange data using the mailbox.
    The following shows the basic form of coding TA_MPRI attribute messages.

    [Message packet for TA_MPRI attribute]

```
/* T_MSG structure, which is defined in the kernel.h*/
typedef struct {
    VP      msghead;       /*RI600V4 management area*/
} T_MSG;

/* T_MSG_PRI structure, which is defined in the kernel.h*/
typedef struct {
    T_MSG   msgque;        /*Message header*/
    PRI     msgpri;        /*Message priority*/
} T_MSG_PRI;

/* Message structure defined by user*/
typedef struct {
    T_MSG_PRI t_msg;       /*T_MSG_PRI structure*/
    B         data[8];     /*User message*/
} USER_MSG;
```

Note 1 In the RI600V4, a message having a smaller priority number is given a higher priority.

Note 2 Values that can be specified as the message priority level are limited to the range defined by Maximum message priority (max_pri) in Mailbox Information (mailbox[])) when the system configuration file is created.

5.5.2 Create mailbox

In the RI600V4, the method of creating a mailbox is limited to “static creation”.
Mailboxes therefore cannot be created dynamically using a method such as issuing a service call from a processing program.
Static mailbox creation means defining of mailboxes using static API “mailbox[]” in the system configuration file.
For details about the static API “mailbox[]”, refer to “19.11 Mailbox Information (mailbox[])”.

5.5.3 Send to mailbox

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

- snd_mbx, isnd_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 these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      mbxid = 1;              /*Declares and initializes variable*/
    T_MSG_PRI       *pk_msg;        /*Declares data structure*/

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

    /* ......... */                 /*Secures memory area (for message)*/
    pk_msg = ...                    /* and set the pointer to pk_msg*/
    /* ......... */                 /*Creates message (contents)*/

    pk_msg->msgpri = 8;             /*Initializes data structure*/

                                    /*Send to mailbox*/
    snd_mbx (mbxid, (T_MSG *) pk_msg);

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

Note 1 Messages are queued to the target mailbox in the order defined by queuing method during configuration (FIFO order or message priority order).

Note 2 For details about the message packet T_MSG and T_MSG_PRI, refer to “5.5.1 Messages”.

5.5.4 Receive from mailbox

A message is received (infinite wait, polling, or with time-out) by issuing the following service call from the processing program.

- rcv_mbx (Wait)
- prcv_mbx, iprcv_mbx (Polling)
- trcv_mbx (Wait with time-out)

- rcv_mbx (Wait)
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 no message could be received from the target mailbox (no messages were queued to the wait queue) when this service call is issued, this service call does not receive messages but queues the invoking task to the target mailbox wait queue and moves it from the RUNNING state to the WAITING state (message reception wait state).
The receiving WAITING state for a mailbox is cancelled in the following cases.

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
A message was transmitted to the target mailbox as a result of issuing isnd_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"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mbxid = 1;              /*Declares and initializes variable*/
    T_MSG   *ppk_msg;               /*Declares data structure*/

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

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

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

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

Note 1 Invoking tasks are queued to the target mailbox wait queue in the order defined during configuration (FIFO order or current priority order).

Note 2 For details about the message packet T_MSG and T_MSG_PRI, refer to "5.5.1 Messages".

- prcv_mbx, iprcv_mbx (Polling)
  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 these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mbxid = 1;              /*Declares and initializes variable*/
    T_MSG   *ppk_msg;               /*Declares data structure*/

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

                                    /*Receive from mailbox*/
    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 T_MSG and T_MSG_PRI, refer to “5.5.1 Messages”.

- trcv_mbx (Wait with time-out)
  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 no message could be received from the target mailbox (no messages were queued to the wait queue) when this service call is issued, this service call does not receive messages but queues the invoking task to the target mailbox wait queue and moves it from the RUNNING state to the WAITING state with time-out (message reception wait state).
  The receiving WAITING state for a mailbox is cancelled in the following cases.

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
A message was transmitted to the target mailbox as a result of issuing isnd_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 time specified by *tmout* has elapsed.	E_TMOUT

The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mbxid = 1;              /*Declares and initializes variable*/
    T_MSG   *ppk_msg;               /*Declares data structure*/
    TMO     tmout = 3600;           /*Declares and initializes variable*/

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

                                    /*Receive from mailbox*/
    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) {
        /* ......... */             /*Time-out processing*/
    }

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

Note 1 Invoking tasks are queued to the target mailbox wait queue in the order defined during configuration (FIFO order or current priority order).

Note 2 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 3 For details about the message packet T_MSG and T_MSG_PRI, refer to "5.5.1 Messages".

5.5.5 Reference mailbox state

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

- ref_mbx, iref_mbx
  Stores mailbox state packet (ID number of the task at the head of the wait queue, start address of the message packet at the head of the wait queue) of the mailbox specified by parameter *mbxid* in the area specified by parameter *pk_rmbx*.
  The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      mbxid = 1;              /*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;        /*Reference start address of the message */
                                    /*packet at the head of the wait queue*/

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

Note For details about the mailbox state packet, refer to “[Mailbox state packet: T_RMBX]”.

CHAPTER 6 EXTENDED SYNCHRONIZATION AND COMMUNICATION FUNCTIONS

This chapter describes the extended synchronization and communication functions performed by the RI600V4.

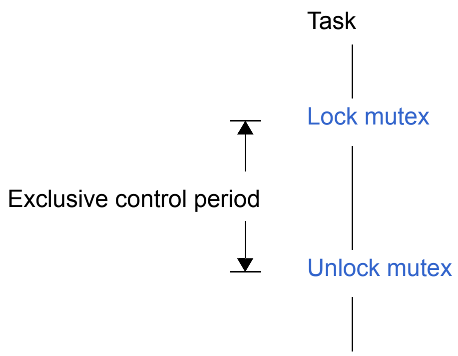
6.1 Outline

The extended synchronization and communication function of the RI600V4 provides Mutexes for implementing exclusive control between tasks, and Message Buffers for transferring messages of he arbitrary size by copying the message.

6.2 Mutexes

Multitask processing requires the function to prevent contentions on using the limited number of resources (A/D converter, coprocessor, files, or the like) simultaneously by tasks operating in parallel (exclusive control function). To resolve such problems, the RI600V4 therefore provides “mutexes”.
The following shows a processing flow when using a mutex.
The mutexes provided in the RI600V4 supports the priority ceiling protocol.

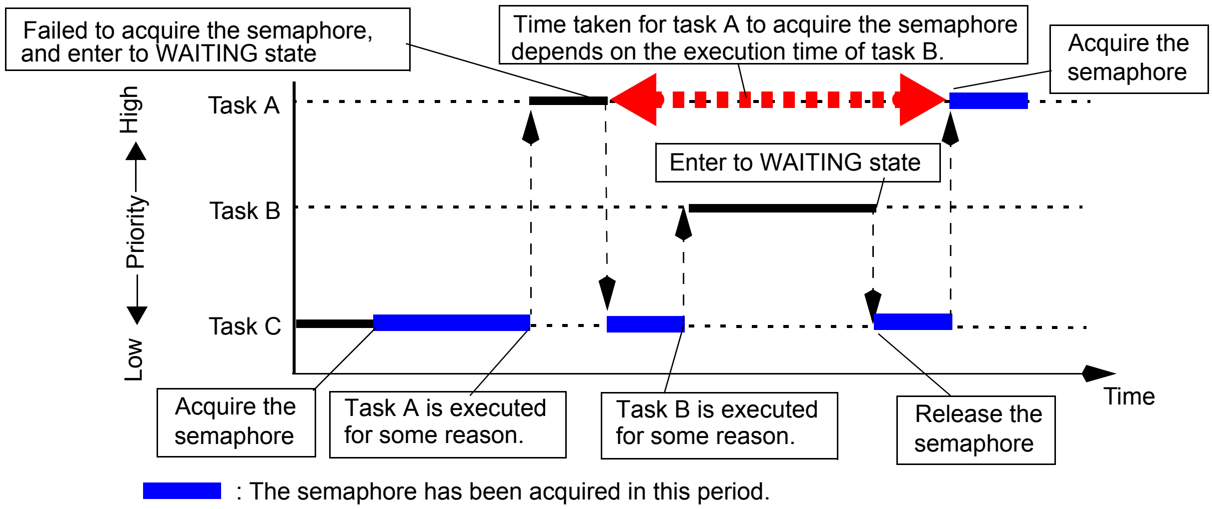
Figure 6-1 Processing Flow (Mutex)



6.2.1 Priority inversion problem

When a semaphore is used for exclusive control of a resource, a problem called priority inversion may arise. This refers to the situation where a task that is not using a resource delays the execution of a task requesting the resource.
Figure 6-2 illustrates this problem. In this figure, tasks A and C are using the same resource, which task B does not use. Task A attempts to acquire a semaphore so that it can use the resource but enters the WAITING state because task C is already using the resource. Task B has a priority higher than task C and lower than task A. Thus, if task B is executed before task C has released the semaphore, release of the semaphore is delayed by the execution of task B. This also delays acquisition of the semaphore by task A. From the viewpoint of task A, a lower-priority task that is not even competing for the resource gets priority over task A.
To avoid this problem, use a mutex instead of a semaphore.

Figure 6-2 Priority Inversion Problem



6.2.2 Current priority and base priority

A task has two priority levels: base priority and current priority. Tasks are scheduled according to current priority.
While a task does not have a mutex locked, its current priority is always the same as its base priority.
When a task locks a mutex, only its current priority is raised to the ceiling priority of the mutex.
When priority-changing service call chg_pri or ichg_pri is issued, both the base priority and current priority are changed if the specified task does not have a mutex locked. When the specified task locks a mutex, only the base priority is changed.
When the specified task has a mutex locked or is waiting to lock a mutex, these service calls returns “E_ILUSE” if a priority higher than the ceiling priority of the mutex is specified.
The current priority can be checked through service call get_pri or iget_pri. And both the current priority and base priority can be referred by ref_tsk or iref_tsk.

6.2.3 Simplified priority ceiling protocol

Original behavior of the priority ceiling protocol is to make the current priority of the task to the highest ceiling priority of mutexes which are locked by the task. This behavior is achieved by controlling the current priority of the task as follows.

- When a task locks a mutex, changes the current priority of the task to the highest ceiling priority of mutexes which are locked by the task.
- When a task unlocks a mutex, <u>changes the current priority of the task to the highest ceiling priority of mutexes which continues to be locked by the task.</u> When there is no mutex locked by the task after unlock, returns the current priority of the task to the base priority.

However, the RI600V4 adopts simplified priority ceiling protocol because of reducing overhead. Therefore, the underlined part is not processed.

6.2.4 Differences from semaphores

The mutex operates similarly to semaphores (binary semaphore) whose the maximum resource count is 1, but they differ in the following points.

- The current priority of the task which locks a mutex raises to the ceiling priority of the mutex until the task unlocks the mutex. As a result, the priority inversion problem is evaded.
  --> The current priority is not changed by using semaphore.

- A locked mutex can be unlocked (equivalent to returning of resources) only by the task that locked the mutex
  --> Semaphores can return resources via any task and handler.

- Unlocking is automatically performed when a task that locked the mutex is terminated (ext_tsk or ter_tsk)
  --> Semaphores do not return resources automatically, so they end with resources acquired.

- Semaphores can manage multiple resources (the maximum resource count can be assigned), but the maximum number of resources assigned to a mutex is fixed to 1.

6.2.5 Create mutex

In the RI600V4, the method of creating a mutex is limited to “static creation”.
Mutexes therefore cannot be created dynamically using a method such as issuing a service call from a processing program.
Static mutex creation means defining of mutexes using static API “mutex[]” in the system configuration file.
For details about the static API “mutex[]”, refer to “19.12 Mutex Information (mutex[])”.

6.2.6 Lock mutex

Mutexes can be locked by issuing the following service call from the processing program.

- loc_mtx (Wait)
- ploc_mtx (Polling)
- tloc_mtx (Wait with time-out)

- loc_mtx (Wait)
This service call locks the mutex specified by parameter *mtxid*.
If the target mutex could not be locked (another task has been locked) when this service call is issued, this service call queues the invoking task to the target mutex wait queue and moves it from the RUNNING state to the WAITING state (mutex wait state).
The WAITING state for a mutex is cancelled in the following cases.

WAITING State for a Mutex Cancel Operation	Return Value
The locked state of the target mutex was cancelled as a result of issuing unl_mtx.	E_OK
The locked state of the target mutex was cancelled as a result of issuing ext_tsk.	E_OK
The locked state of the target mutex was cancelled as a result of issuing ter_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

When the mutex is locked, this service call changes the current priority of the invoking task to the ceiling priority of the target mutex. However, this service call does not change the current priority when the invoking task has locked other mutexes and the ceiling priority of the target mutex is lower than or equal to the ceiling priority of the locked mutexes.
The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mtxid = 8;              /*Declares and initializes variable*/

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

    ercd = loc_mtx (mtxid);         /*Lock mutex*/

    if (ercd == E_OK) {
        /* ......... */             /*Locked state*/

        unl_mtx (mtxid);            /*Unlock mutex*/
    } else if (ercd == E_RLWAI) {
        /* ......... */             /*Forced termination processing*/
    }

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

Note 1 Invoking tasks are queued to the target mutex wait queue in the priority order. Among tasks with the same priority, they are queued in FIFO order.

Note 2 This service call returns “E_ILUSE” if this service call is re-issued for the mutex that has been locked by the invoking task (multiple-locking of mutex).

- ploc_mtx (Polling)
  This service call locks the mutex specified by parameter *mtxid*.
  If the target mutex could not be locked (another task has been locked) when this service call is issued but "E_TMOUT" is returned.
  When the mutex is locked, this service call changes the current priority of the invoking task to the ceiling priority of the target mutex. However, the this service call does not change the current priority when the invoking task has locked other mutexes and the ceiling priority of the target mutex is lower than or equal to the ceiling priority of the locked mutexes.
  The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mtxid = 8;              /*Declares and initializes variable*/

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

    ercd = ploc_mtx (mtxid);        /*Lock mutex*/

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

        unl_mtx (mtxid);            /*Unlock mutex*/
    } else if (ercd == E_TMOUT) {
        /* ......... */             /*Polling failure processing*/
    }

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

Note This service call returns "E_ILUSE" if this service call is re-issued for the mutex that has been locked by the invoking task (multiple-locking of mutex).

- tloc_mtx (Wait with time-out)
  This service call locks the mutex specified by parameter *mtxid*.
  If the target mutex could not be locked (another task has been locked) when this service call is issued, this service call queues the invoking task to the target mutex wait queue and moves it from the RUNNING state to the WAITING state with time-out (mutex wait state).
  The WAITING state for a mutex is cancelled in the following cases.

WAITING State for a Mutex Cancel Operation	Return Value
The locked state of the target mutex was cancelled as a result of issuing unl_mtx.	E_OK
The locked state of the target mutex was cancelled as a result of issuing ext_tsk.	E_OK
The locked state of the target mutex was cancelled as a result of issuing ter_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 time specified by *tmout* has elapsed.	E_TMOUT

When the mutex is locked, this service call changes the current priority of the invoking task to the ceiling priority of the target mutex. However, the this service call does not change the current priority when the invoking task has locked other mutexes and the ceiling priority of the target mutex is lower than or equal to the ceiling priority of the locked mutexes.
The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mtxid = 8;              /*Declares and initializes variable*/
    TMO     tmout = 3600;           /*Declares and initializes variable*/

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

    ercd = tloc_mtx (mtxid, tmout); /*Lock mutex*/

    if (ercd == E_OK) {
        /* ......... */             /*Locked state*/

        unl_mtx (mtxid);            /*Unlock mutex*/
    } else if (ercd == E_RLWAI) {
        /* ......... */             /*Forced termination processing*/
    } else if (ercd == E_TMOUT) {
        /* ......... */             /*Time-out processing*/
    }

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

Note 1 Invoking tasks are queued to the target mutex wait queue in the priority order. Among tasks with the same priority, they are queued in FIFO order.

Note 2 This service call returns “E_ILUSE” if this service call is re-issued for the mutex that has been locked by the invoking task (multiple-locking of mutex).

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

6.2.7 Unlock mutex

The mutex locked state can be cancelled by issuing the following service call from the processing program.

- unl_mtx
  This service call unlocks the locked mutex specified by parameter *mtxid*.
  If a task has been queued to the target mutex wait queue when this service call is issued, mutex lock processing is performed by the task (the first task in the wait queue) immediately after mutex unlock processing.
  As a result, the task is unlinked from the wait queue and moves from the WAITING state (mutex wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state. And this service call changes the current priority of the task to the ceiling priority of the target mutex. However, this service call does not change the current priority when the task has locked other mutexes and the ceiling priority of the target mutex is lower than or equal to the ceiling priority of the locked mutexes.
  The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mtxid = 8;              /*Declares and initializes variable*/

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

    ercd = loc_mtx (mtxid);         /*Lock mutex*/

    if (ercd == E_OK) {
        /* ......... */             /*Locked state*/

        unl_mtx (mtxid);            /*Unlock mutex*/
    } else if (ercd == E_RLWAI) {
        /* ......... */             /*Forced termination processing*/
    }

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

Note 1 A locked mutex can be unlocked only by the task that locked the mutex.
If this service call is issued for a mutex that was not locked by the invoking task, no processing is performed but “E_ILUSE” is returned.

Note 2 When terminating a task, the mutexes which are locked by the terminated task are unlocked.

6.2.8 Reference mutex state

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

- ref_mtx,
  This service call stores the detailed information of the mutex specified by parameter *mtxid* (existence of locked mutexes, waiting tasks, etc.) into the area specified by parameter *pk_rmtx*.
  The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      mtxid = 1;              /*Declares and initializes variable*/
    T_RMTX  pk_rmtx;                /*Declares data structure*/
    ID      htskid;                 /*Declares variable*/
    ID      wtskid;                 /*Declares variable*/

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

    ref_mtx (mbxid, &pk_rmtx);      /*Reference mutex state*/

    htskid = pk_rmtx.htskid;        /*Acquires existence of locked mutexes*/
    wtskid = pk_rmtx.wtskid;        /*Reference ID number of the task at the */
                                    /*head of the wait queue*/

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

Note For details about the mutex state packet, refer to "[Mutex state packet: T_RMTX]".

6.3 Message Buffers

Multitask processing requires the inter-task communication function (message transfer function) that reports the processing result of a task to another task. The RI600V4 therefore provides the message buffers for copying and transferring the arbitrary size of message.
The following shows a processing flow when using a message buffer.

Figure 6-3 Processing Flow (Message buffer)

6.3.1 Create message buffer

In the RI600V4, the method of creating a message buffer is limited to “static creation”.
Message buffers therefore cannot be created dynamically using a method such as issuing a service call from a processing program.
Static message buffer creation means defining of message buffers using static API “message_buffer[]” in the system configuration file.
For details about the static API “message_buffer[]”, refer to “19.13 Message Buffer Information (message_buffer[])”.

6.3.2 Send to message buffer

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

- snd_mbf (Wait)
- psnd_mbf, ipsnd_mbf (Polling)
- tsnd_mbf (Wait with time-out)

- snd_mbf (Wait)
  This service call processes as follows according to the situation of the message buffer specified by the parameter *mbfid*.

  - There is a task in the reception wait queue.
    This service call transfers the message specified by parameter *msg* to the task in the top of the reception wait queue. As a result, the task is unlinked from the reception wait queue and moves from the WAITING state (message reception wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

  - There is no task neither in the reception wait queue and transmission wait queue and there is available space in the message buffer.
    This service call stores the message specified by parameter *msg* to the message buffer. As a result, the size of available space in the target message buffer decreases by the amount calculated by the following expression.

    The amount of decrease = up4(*msgsz*) + VTSZ_MBFTBL

  - There is no task neither in the reception wait queue and transmission wait queue and there is no available space in the message buffer, or there is a task in the transmission wait queue.
    This service call queues the invoking task to the transmission wait queue of the target message buffer and moves it from the RUNNING state to the WAITING state (message transmission wait state).
    The sending WAITING state for a message buffer is cancelled in the following cases.

Sending WAITING State for a Message Buffer Cancel Operation	Return Value
Available space was secured in the message buffer area as a result of issuing rcv_mbf.	E_OK
Available space was secured in the message buffer area as a result of issuing prcv_mbf.	E_OK
Available space was secured in the message buffer area as a result of issuing trcv_mbf.	E_OK
The task at the top of the transmission wait queue was forcedly released from waiting by following either. - Forced release from waiting (accept rel_wai while waiting). - Forced release from waiting (accept irel_wai while waiting). - Forced release from waiting (accept ter_tsk while waiting). - The time specified by *tmout* for tsnd_mbf 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 message buffer is reset as a result of issuing vrst_mbf.	EV_RST

The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mbfid = 1;              /*Declares and initializes variable*/
    B       msg[] = {1,2,3};        /*Declares and initializes variable*/
    UINT    msgsz = sizeof( msg );  /*Declares and initializes variable*/

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

    ercd = snd_mbf (mbfid, (VP)msg, msgsz);   /*Send to message buffer*/

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

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

Note 1 Message is written to the message buffer area in the order of the message transmission request.

Note 2 Invoking tasks are queued to the transmission wait queue of the target message buffer in the FIFO order.

- psnd_mbf, ipsnd_mbf (Polling)
  This service call processes as follows according to the situation of the message buffer specified by the parameter *mbfid*.
  - There is a task in the reception wait queue.
    This service call transfers the message specified by parameter *msg* to the task in the top of the reception wait queue. As a result, the task is unlinked from the reception wait queue and moves from the WAITING state (message reception wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  - There is no task neither in the reception wait queue and transmission wait queue and there is available space in the message buffer.
    This service call stores the message specified by parameter *msg* to the message buffer. As a result, the size of available space in the target message buffer decreases by the amount calculated by the following expression.

    The amount of decrease = up4(*msgsz*) + VTSZ_MBFTBL
  - There is no task neither in the reception wait queue and transmission wait queue and there is no available space in the message buffer, or there is a task in the transmission wait queue.
    This service call returns “E_TMOUT”.

The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mbfid = 1;              /*Declares and initializes variable*/
    B       msg[] = {1,2,3};        /*Declares and initializes variable*/
    UINT    msgsz = sizeof( msg );  /*Declares and initializes variable*/

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

    ercd = psnd_mbf (mbfid, (VP)msg, msgsz); /*Send to message buffer*/

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

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

Note Message is written to the message buffer area in the order of the message transmission request.

- tsnd_mbf (Wait with time-out)
  This service call processes as follows according to the situation of the message buffer specified by the parameter *mbfid*.

  - There is a task in the reception wait queue.
    This service call transfers the message specified by parameter *msg* to the task in the top of the reception wait queue. As a result, the task is unlinked from the reception wait queue and moves from the WAITING state (message reception wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

  - There is no task neither in the reception wait queue and transmission wait queue and there is available space in the message buffer.
    This service call stores the message specified by parameter *msg* to the message buffer. As a result, the size of available space in the target message buffer decreases by the amount calculated by the following expression.

    The amount of decrease = up4(*msgsz*) + VTSZ_MBFTBL

  - There is no task neither in the reception wait queue and transmission wait queue and there is no available space in the message buffer, or there is a task in the transmission wait queue.
    This service call queues the invoking task to the transmission wait queue of the target message buffer and moves it from the RUNNING state to the WAITING state with time (message transmission wait state).
    The sending WAITING state for a message buffer is cancelled in the following cases.

Sending WAITING State for a Message Buffer Cancel Operation	Return Value
Available space was secured in the message buffer area as a result of issuing rcv_mbf.	E_OK
Available space was secured in the message buffer area as a result of issuing prcv_mbf.	E_OK
Available space was secured in the message buffer area as a result of issuing trcv_mbf.	E_OK
The task at the top of the transmission wait queue was forcedly released from waiting by following either. - Forced release from waiting (accept rel_wai while waiting). - Forced release from waiting (accept irel_wai while waiting). - Forced release from waiting (accept ter_tsk while waiting). - The time specified by *tmout* for tsnd_mbf 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 message buffer is reset as a result of issuing vrst_mbf.	EV_RST
The time specified by *tmout* has elapsed.	E_TMOUT

The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mbfid = 1;              /*Declares and initializes variable*/
    B       msg[] = {1,2,3};        /*Declares and initializes variable*/
    TMO     tmout = 3600;           /*Declares and initializes variable*/

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

    ercd = tsnd_mbf (mbfid, (VP)msg, msgsz, tmout); /*Send to message buffer*/

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

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

Note 1 Message is written to the message buffer area in the order of the message transmission request.

Note 2 Invoking tasks are queued to the transmission wait queue of the target message buffer in the FIFO order.

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

6.3.3 Receive from message buffer

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

- rcv_mbf (Wait)
- prcv_mbf (Polling)
- trcv_mbf (Wait with time-out)

- rcv_mbf (Wait)
  This service call processes as follows according to the situation of the message buffer specified by the parameter *mbfid*.
  - There is a message in the message buffer.
    This service call takes out the oldest message from the message buffer and stores the message to the area specified by *msg* and return the size of the message. As a result, the size of available space in the target message buffer increases by the amount calculated by the following expression.

    The amount of increase = up4(Return value) + VTSZ_MBFTBL

    In addition, this service call repeats the following processing until task in the transmission wait queue is lost or it becomes impossible to store the message in the message buffer.
    - When there is a task in the transmission wait queue and there is available space in the message buffer for the message specified by the task in the top of the transmission wait queue, the task is unlinked from the transmission wait queue and moves from the WAITING state (message transmission wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state. As a result, the size of available space in the target message buffer decreases by the amount calculated by the following expression.

      The amount of decrease =up4(The message size sent by the task) + VTSZ_MBFTBL
  - There is no message in the message buffer and there is a task in the transmission wait queue.
    This service call stores the message specified by the task in the top of the transmission wait queue to the area pointed by the parameter *msg*. As a result, the task is unlinked from the transmission wait queue and moves from the WAITING state (message transmission wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
    Note, this situation is caused only when the size of the message buffer is 0.
  - There is no message in the message buffer and there is no task in the transmission wait queue.
    This service call queues the invoking task to the reception wait queue of the target message buffer and moves it from the RUNNING state to the WAITING state (message reception wait state).
    The receiving WAITING state for a message buffer is cancelled in the following cases.

Receiving WAITING State for a Message Buffer Cancel Operation	Return Value
Message was sent to the message buffer area as a result of issuing snd_mbf.	E_OK
Message was sent to the message buffer area as a result of issuing psnd_mbf.	E_OK
Message was sent to the message buffer area as a result of issuing ipsnd_mbf.	E_OK
Message was sent to the message buffer area as a result of issuing tsnd_mbf.	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"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mbfid = 1;              /*Declares and initializes variable*/
    B       msg[16];                /*Declares variable (maximum message size)*/

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

    ercd = rcv_mbf (mbfid, (VP)msg); /*Receive from message buffer */

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

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

Note 1 The Maximum message size (max_msgsz) is defined during configuration. The size of the area pointed by msg must be larger than or equal to the maximum message size.

Note 2 Invoking tasks are queued to the reception wait queue of the target message buffer in the order of the message reception request.

- prcv_mbf (Polling)
  This service call processes as follows according to the situation of the message buffer specified by the parameter *mbfid*.
  - There is a message in the message buffer.
    This service call takes out the oldest message from the message buffer and stores the message to the area specified by *msg* and return the size of the message. As a result, the size of available space in the target message buffer increases by the amount calculated by the following expression.

    The amount of increase = up4(Return value) + VTSZ_MBFTBL

    In addition, this service call repeats the following processing until task in the transmission wait queue is lost or it becomes impossible to store the message in the message buffer.
    - When there is a task in the transmission wait queue and there is available space in the message buffer for the message specified by the task in the top of the transmission wait queue, the task is unlinked from the transmission wait queue and moves from the WAITING state (message transmission wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state. As a result, the size of available space in the target message buffer decreases by the amount calculated by the following expression.

      The amount of decrease =up4(The message size sent by the task) + VTSZ_MBFTBL
  - There is no message in the message buffer and there is a task in the transmission wait queue.
    This service call stores the message specified by the task in the top of the transmission wait queue to the area pointed by the parameter *msg*. As a result, the task is unlinked from the transmission wait queue and moves from the WAITING state (message transmission wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
    Note, this situation is caused only when the size of the message buffer is 0.
  - There is no message in the message buffer and there is no task in the transmission wait queue.
    This service call returns “E_TMOUT”.

The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mbfid = 1;              /*Declares and initializes variable*/
    B       msg[16];                /*Declares variable (maximum message size)*/

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

    ercd = prcv_mbf (mbfid, (VP)msg); /*Receive from message buffer */

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

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

Note The Maximum message size (max_msgsz) is defined during configuration. The size of the area pointed by msg must be larger than or equal to the maximum message size.

- trcv_mbf (Wait with time-out)
  This service call processes as follows according to the situation of the message buffer specified by the parameter *mbfid*.

  - There is a message in the message buffer.
    This service call takes out the oldest message from the message buffer and stores the message to the area specified by *msg* and return the size of the message. As a result, the size of available space in the target message buffer increases by the amount calculated by the following expression.

    The amount of increase = up4(Return value) + VTSZ_MBFTBL

    In addition, this service call repeats the following processing until task in the transmission wait queue is lost or it becomes impossible to store the message in the message buffer.

    - When there is a task in the transmission wait queue and there is available space in the message buffer for the message specified by the task in the top of the transmission wait queue, the task is unlinked from the transmission wait queue and moves from the WAITING state (message transmission wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state. As a result, the size of available space in the target message buffer decreases by the amount calculated by the following expression.

      The amount of decrease =up4(The message size sent by the task) + VTSZ_MBFTBL

  - There is no message in the message buffer and there is a task in the transmission wait queue.
    This service call stores the message specified by the task in the top of the transmission wait queue to the area pointed by the parameter *msg*. As a result, the task is unlinked from the transmission wait queue and moves from the WAITING state (message transmission wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
    Note, this situation is caused only when the size of the message buffer is 0.

  - There is no message in the message buffer and there is no task in the transmission wait queue.
    This service call queues the invoking task to the reception wait queue of the target message buffer and moves it from the RUNNING state to the WAITING state with time (message reception wait state).
    The receiving WAITING state for a message buffer is cancelled in the following cases.

Receiving WAITING State for a Message Buffer Cancel Operation	Return Value
Message was sent to the message buffer area as a result of issuing snd_mbf.	E_OK
Message was sent to the message buffer area as a result of issuing psnd_mbf.	E_OK
Message was sent to the message buffer area as a result of issuing ipsnd_mbf.	E_OK
Message was sent to the message buffer area as a result of issuing tsnd_mbf.	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 time specified by *tmout* has elapsed.	E_TMOUT

The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mbfid = 1;              /*Declares and initializes variable*/
    B       msg[16];                /*Declares variable (maximum message size)*/
    TMO     tmout = 3600;           /*Declares and initializes variable*/

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

    ercd = trcv_mbf (mbfid, (VP)msg, tmout ); /*Receive from message buffer */

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

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

Note 1 The Maximum message size (max_msgsz) is defined during configuration. The size of the area pointed by msg must be larger than or equal to the maximum message size.

Note 2 Invoking tasks are queued to the reception wait queue of the target message buffer in the order of the message reception request.

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

6.3.4 Reference message buffer state

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

- ref_mbf, iref_mbf
  These service calls store the detailed information of the message buffer (existence of waiting tasks, available buffer size, etc.) specified by parameter *mbfid* into the area specified by parameter *pk_rmbf*.
  The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      mbfid = 1;              /*Declares and initializes variable*/
    T_RMBF  pk_rmbf;                /*Declares message structure*/
    ID      stskid;                 /*Declares variable*/
    ID      rtskid;                 /*Declares variable*/
    UINT    smsgcnt;                /*Declares variable*/
    SIZE    fmbfsz;                 /*Declares variable*/

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

    ref_mbf (mbfid, &pk_rmbf);       /*Reference message buffer state*/

    stskid = pk_rmbf.stskid;        /*Acquires existence of tasks waiting for */
                                    /*message transmission*/
    rtskid = pk_rmbf.rtskid;        /*Acquires existence of tasks waiting for */
                                    /*message reception*/
    smsgcnt = pk_rmbf.smsgcnt;      /*Acquires the number of message in */
                                    /*message buffer*/
    fmbfsz = pk_rmbf.fmbfsz;        /*Acquires the available buffer size */

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

Note For details about the message buffer state packet, refer to "[Message buffer state packet: T_RMBF]".

CHAPTER 7 MEMORY POOL MANAGEMENT FUNCTIONS

This chapter describes the memory pool management functions performed by the RI600V4.

7.1 Outline

The RI600V4 provides “Fixed-Sized Memory Pools” and “Variable-Sized Memory Pools” as dynamic memory allocation function.
In the fixed-sized memory pool, the size of memory that can use is fixation, but the over-head to acquire/release memory is short.
On the other hand, in the variable-sized memory pool, memory of the arbitrary size can be used, but the over-head to acquire/release memory is longer than the fixed-sized memory pool. And fragmentation of the memory pool area may occur.

7.2 Fixed-Sized Memory Pools

When a dynamic memory manipulation request is issued from a processing program in the RI600V4, the fixed-sized memory pool is provided as a usable memory area.
Dynamic memory manipulation of the fixed-size memory pool is executed in fixed size memory block units.

7.2.1 Create fixed-sized memory pool

In the RI600V4, the method of creating a fixed-sized memory pool is limited to “static creation”.
Fixed-sized memory pools therefore cannot be created dynamically using a method such as issuing a service call from a processing program.
Static fixed-size memory pool creation means defining of fixed-size memory pools using static API “memorypool[]” in the system configuration file.
For details about the static API “memorypool[]”, refer to “19.14 Fixed-sized Memory Pool Information (memorypool[])”.

7.2.2 Acquire fixed-sized memory block

A fixed-sized memory block is acquired (waiting forever, polling, or with time-out) by issuing the following service call from the processing program.

- get_mpf (Wait)
- pget_mpf, ipget_mpf (Polling)
- tget_mpf (Wait with time-out)

The RI600V4 does not perform memory clear processing when a fixed-sized memory block is acquired. The contents of the acquired fixed-size memory block are therefore undefined.

- get_mpf (Wait)
  This service call acquires the fixed-sized 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 no fixed-size memory blocks could be acquired from the target fixed-size memory pool (no available fixed-size memory blocks exist) when this service call is issued, this service call does not acquire the fixed-size memory block but queues the invoking task to the target fixed-size memory pool wait queue and moves it from the RUNNING state to the WAITING state (fixed-size memory block acquisition wait state).
  The WAITING state for a fixed-sized memory block is cancelled in the following cases.

WAITING State for a Fixed-sized Memory Block Cancel Operation	Return Value
A fixed-sized memory block was returned to the target fixed-sized memory pool as a result of issuing rel_mpf.	E_OK
A fixed-sized memory block was returned to the target fixed-sized memory pool as a result of issuing irel_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 fixed-sized memory pool is reset as a result of issuing vrst_mpf.	EV_RST

The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mpfid = 1;              /*Declares and initializes variable*/
    VP      p_blk;                  /*Declares variable*/

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

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

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

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

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

Note 1 Invoking tasks are queued to the target fixed-size memory pool wait queue in the order defined during configuration (FIFO order or current priority order).

Note 2 The contents of the block are undefined.

Note 3 The boundary alignment for the memory blocks acquired is 1. If memory blocks need to be acquired with a larger boundary alignment than that, observe the following:

- Set The size of the fixed-sized memory block (siz_block) in Fixed-sized Memory Pool Information (memorypool[]) to multiple of the desired boundary alignment.
- Specify unique section name to the Section name assigned to the memory pool area (section) in Fixed-sized Memory Pool Information (memorypool[]) and locate the section to the address of the desired boundary alignment when linking.

- pget_mpf, ipget_mpf (Polling)
  This service call acquires the fixed-sized 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 fixed-sized memory block could not be acquired from the target fixed-sized memory pool (no available fixed-sized memory blocks exist) when this service call is issued, fixed-sized memory block acquisition processing is not performed but “E_TMOUT” is returned.
  The following describes an example for coding these service calls.

```
#include    "kernel.h"                /*Standard header file definition*/
#include    "kernel_id.h"             /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER      ercd;                     /*Declares variable*/
    ID      mpfid = 1;                /*Declares and initializes variable*/
    VP      p_blk;                    /*Declares variable*/

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

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

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

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

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

Note 1 The contents of the block are undefined.

Note 2 The boundary alignment for the memory blocks acquired is 1. If memory blocks need to be acquired with a larger boundary alignment than that, observe the following:

- Set The size of the fixed-sized memory block (siz_block) in Fixed-sized Memory Pool Information (memorypool[]) to multiple of the desired boundary alignment.
- Specify unique section name to the Section name assigned to the memory pool area (section) in Fixed-sized Memory Pool Information (memorypool[]) and locate the section to the address of the desired boundary alignment when linking.

- tget_mpf (Wait with time-out)
  This service call acquires the fixed-sized 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 no fixed-size memory blocks could be acquired from the target fixed-size memory pool (no available fixed-size memory blocks exist) when this service call is issued, this service call does not acquire the fixed-size memory block but queues the invoking task to the target fixed-size memory pool wait queue and moves it from the RUNNING state to the WAITING state with time-out (fixed-size memory block acquisition wait state).
  The WAITING state for a fixed-sized memory block is cancelled in the following cases.

WAITING State for a Fixed-sized Memory Block Cancel Operation	Return Value
A fixed-sized memory block was returned to the target fixed-sized memory pool as a result of issuing rel_mpf.	E_OK
A fixed-sized memory block was returned to the target fixed-sized memory pool as a result of issuing irel_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 fixed-sized memory pool is reset as a result of issuing vrst_mpf.	EV_RST
The time specified by *tmout* has elapsed.	E_TMOUT

The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

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

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

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

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

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

Note 1 Invoking tasks are queued to the target fixed-size memory pool wait queue in the order defined during configuration (FIFO order or current priority order).

Note 2 The contents of the block are undefined.

Note 3 The boundary alignment for the memory blocks acquired is 1. If memory blocks need to be acquired with a larger boundary alignment than that, observe the following:

- Set The size of the fixed-sized memory block (siz_block) in Fixed-sized Memory Pool Information (memorypool[]) to multiple of the desired boundary alignment.
- Specify unique section name to the Section name assigned to the memory pool area (section) in Fixed-sized Memory Pool Information (memorypool[]) and locate the section to the address of the desired boundary alignment when linking.

Note 4 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.

7.2.3 Release fixed-sized memory block

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

- rel_mpf, irel_mpf
  This service call returns the fixed-sized 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, fixed-sized memory block return processing is not performed but fixed-sized 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 these service calls.

```
#include    "kernel.h"                /*Standard header file definition*/
#include    "kernel_id.h"             /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER      ercd;                     /*Declares variable*/
    ID      mpfid = 1;                /*Declares and initializes variable*/
    VP      blk;                      /*Declares variable*/

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

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

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

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

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

7.2.4 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, iref_mpf
  Stores fixed-sized memory pool state packet (ID number of the task at the head of the wait queue, number of free memory blocks, etc.) 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 these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      mpfid = 1;              /*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 "[Fixed-sized memory pool state packet: T_RMPF]".

7.3 Variable-Sized Memory Pools

When a dynamic memory manipulation request is issued from a processing program in the RI600V4, the variable-sized memory pool is provided as a usable memory area.
Dynamic memory manipulation for variable-size memory pools is performed in the units of the specified variable-size memory block size.

7.3.1 Create variable-sized memory pool

In the RI600V4, the method of creating a variable-sized memory pool is limited to “static creation”.
Variable-sized memory pools therefore cannot be created dynamically using a method such as issuing a service call from a processing program.
Static variable-size memory pool creation means defining of variable-size memory pools using static API “variable_memorypool[]” in the system configuration file.
For details about the static API “variable_memorypool[]”, refer to “19.15 Variable-sized Memory Pool Information (variable_memorypool[])”

7.3.2 Size of Variable-sized memory block

In the current implementation of the RI600V4, the size of the variable-sized memory block to be acquired is selected from 12 (in maximum) kinds of variations. This variations are selected from 24 kinds of inside decided beforehand according to Upper limit of the variable-sized memory block (max_memsize) in Variable-sized Memory Pool Information (variable_memorypool[]). Table 7-1 shows variation of memory block size. Note, this behavior may be changed in the future version.

Table 7-1 Variation of memory block size

No,	Size of memory block (Hexadecimal)	Example-1 max_memsize = 0x100	Example-1 max_memsize = 0x20000
1	12 (0xC)	Used	-
2	36 (0x24)	Used	-
3	84 (0x54)	Used	Used
4	180 (0xB4)	Used	Used
5	372 (0x174)	-	Used
6	756 (0x2F4)	-	Used
7	1524 (0x5F4)	-	Used
8	3060 (0xBF4)	-	Used
9	6132 (0x17F4)	-	Used
10	12276 (0x2FF4)	-	Used
11	24564 (0x5FF4)	-	Used
12	49140 (0xBFF4)	-	Used
13	98292 (0x17FF4)	-	Used
14	196596 (0x2FFF4)	-	Used
15	393204 (0x5FFF4)	-	-
16	786420 (0xBFFF4)	-	-
17	1572852 (0x17FFF4)	-	-
18	3145716 (0x2FFFF4)	-	-
19	6291444 (0x5FFFF4)	-	-
20	12582900 (0xBFFFF4)	-	-
21	25165812 (0x17FFFF4)	-	-
22	50331636 (0x2FFFFF4)	-	-
23	100663284 (0x5FFFFF4)	-	-
24	201326580 (0xBFFFFF4)	-	-

7.3.3 Acquire variable-sized memory block

A variable-sized memory block is acquired (waiting forever, polling, or with time-out) by issuing the following service call from the processing program.

- get_mpl (Wait)
- pget_mpl, ipget_mpl (Polling)
- tget_mpl (Wait with time-out)

The RI600V4 does not perform memory clear processing when a variable-sized memory block is acquired. The contents of the acquired variable-size memory block are therefore undefined.

- get_mpl (Wait)
  This service call acquires a variable-size memory block of the size specified by parameter blksz from the variable-size memory pool specified by parameter *mplid*, and stores its start address into the area specified by parameter *p_blk*.
  If no variable-size memory blocks could be acquired from the target variable-size memory pool (no successive areas equivalent to the requested size were available) when this service call is issued, this service call does not acquire variable-size memory blocks but queues the invoking task to the target variable-size memory pool wait queue and moves it from the RUNNING state to the WAITING state (variable-size memory block acquisition wait state).
  The WAITING state for a variable-sized memory block is cancelled in the following cases.

WAITING State for a Variable-sized Memory Block Cancel Operation	Return Value
The variable-size memory block that satisfies the requested size was returned to the target variable-size memory pool as a result of issuing rel_mpl.	E_OK
The task at the top of the transmission wait queue was forcedly released from waiting by following either. - Forced release from waiting (accept rel_wai while waiting). - Forced release from waiting (accept irel_wai while waiting). - Forced release from waiting (accept ter_tsk while waiting). - The time specified by *tmout* for tget_mpl 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 variable-sized memory pool is reset as a result of issuing vrst_mpl.	EV_RST

The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mplid = 1;              /*Declares and initializes variable*/
    UINT    blksz = 256;            /*Declares and initializes variable*/
    VP      p_blk;                  /*Declares variable*/

    /* ......... */
                                    /*Acquire variable-sized memory block */
    ercd = get_mpl (mplid, blksz, &p_blk);

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

        rel_mpl (mplid, p_blk);     /*Release variable-sized memory block*/
    } else if (ercd == E_RLWAI) {
        /* ......... */             /*Forced termination processing*/
    }

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

Note 1 For the size of the memory block, refer to "7.3.2 Size of Variable-sized memory block".

Note 2 Invoking tasks are queued to the target variable-size memory pool wait queue in the FIFO order.

Note 3 The contents of the block are undefined.

Note 4 The alignment number of memory blocks is 1. To enlarge the alignment number to 4, specify unique section to Section name assigned to the memory pool area (mpl_section) in Variable-sized Memory Pool Information (variable_memorypool[]) and locate the section to 4-bytes boundary address when linking.

- pget_mpl, ipget_mpl (Polling)
  This service call acquires a variable-size memory block of the size specified by parameter *blksz* from the variable-size memory pool specified by parameter *mplid*, and stores its start address into the area specified by parameter *p_blk*.
  If no variable-size memory blocks could be acquired from the target variable-size memory pool (no successive areas equivalent to the requested size were available) when this service call is issued, this service call does not acquire variable-size memory block but returns “E_TMOUT”.
  The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mplid = 1;              /*Declares and initializes variable*/
    UINT    blksz = 256;            /*Declares and initializes variable*/
    VP      p_blk;                  /*Declares variable*/

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

                                    /*Acquire variable-sized memory block*/
    ercd = pget_mpl (mplid, blksz, &p_blk);

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

        rel_mpl (mplid, p_blk);     /*Release variable-sized memory block*/
    } else if (ercd == E_TMOUT) {
        /* ......... */             /*Polling failure processing*/
    }

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

Note 1 For the size of the memory block, refer to “7.3.2 Size of Variable-sized memory block”.

Note 2 The contents of the block are undefined.

Note 3 The alignment number of memory blocks is 1. To enlarge the alignment number to 4, specify unique section to Section name assigned to the memory pool area (mpl_section) in Variable-sized Memory Pool Information (variable_memorypool[]) and locate the section to 4-bytes boundary address when linking.

- tget_mpl (Wait with time-out)
  This service call acquires a variable-size memory block of the size specified by parameter *blksz* from the variable-size memory pool specified by parameter *mplid*, and stores its start address into the area specified by parameter *p_blk*.
  If no variable-size memory blocks could be acquired from the target variable-size memory pool (no successive areas equivalent to the requested size were available) when this service call is issued, this service call does not acquire variable-size memory blocks but queues the invoking task to the target variable-size memory pool wait queue and moves it from the RUNNING state to the WAITING state with time-out (variable-size memory block acquisition wait state).
  The WAITING state for a variable-sized memory block is cancelled in the following cases.

WAITING State for a Variable-sized Memory Block Cancel Operation	Return Value
The variable-size memory block that satisfies the requested size was returned to the target variable-size memory pool as a result of issuing rel_mpl.	E_OK
The task at the top of the transmission wait queue was forcedly released from waiting by following either. - Forced release from waiting (accept rel_wai while waiting). - Forced release from waiting (accept irel_wai while waiting). - Forced release from waiting (accept ter_tsk while waiting). - The time specified by *tmout* for tget_mpl 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 variable-sized memory pool is reset as a result of issuing vrst_mpl.	EV_RST
The time specified by *tmout* has elapsed.	E_TMOUT

The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void
task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mplid = 1;              /*Declares and initializes variable*/
    UINT    blksz = 256;            /*Declares and initializes variable*/
    VP      p_blk;                  /*Declares variable*/
    TMO     tmout = 3600;           /*Declares and initializes variable*/

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

                                    /*Acquire variable-sized memory block*/
    ercd = tget_mpl (mplid, blksz, &p_blk, tmout);

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

        rel_mpl (mplid, p_blk ;     /*Release variable-sized memory block*/
    } else if (ercd == E_RLWAI) {
        /* ......... */             /*Forced termination processing*/
    } else if (ercd == E_TMOUT) {
        /* ......... */             /*Time-out processing*/
    }

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

Note 1 For the size of the memory block, refer to "7.3.2 Size of Variable-sized memory block".

Note 2 Invoking tasks are queued to the target variable-size memory pool wait queue in the FIFO order.

Note 3 The contents of the block are undefined.

Note 4 The alignment number of memory blocks is 1. To enlarge the alignment number to 4, specify unique section to Section name assigned to the memory pool area (mpl_section) in Variable-sized Memory Pool Information (variable_memorypool[]) and locate the section to 4-bytes boundary address when linking.

Note 5 TMO_FEVR is specified for wait time *tmout*, processing equivalent to get_mpl will be executed. When TMO_POL is specified, processing equivalent to pget_mpl will be executed.

7.3.4 Release variable-sized memory block

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

- rel_mpl
  This service call returns the variable-sized memory block specified by parameter *blk* to the variable-sized memory pool specified by parameter *mplid*.
  After returning the variable-size memory blocks, these service calls check the tasks queued to the target variable-size memory pool wait queue from the top, and assigns the memory if the size of memory requested by the wait queue is available. This operation continues until no tasks queued to the wait queue remain or no memory space is available. As a result, the task that acquired the memory is unlinked from the queue and moved from the WAITING state (variable-size memory block acquisition wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  The following describes an example for coding these service calls.

```
#include    "kernel.h"                /*Standard header file definition*/
#include    "kernel_id.h"             /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ER      ercd;                     /*Declares variable*/
    ID      mplid = 1;                /*Declares and initializes variable*/
    UINT    blksz = 256;              /*Declares and initializes variable*/
    VP      blk;                      /*Declares variable*/

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

                                      /*Acquire variable-sized memory block*/
    ercd = get_mpl (mplid, blksz, &blk);

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

        rel_mpl (mplid, blk);         /*Release variable-sized memory block*/
    } else if (ercd == E_RLWAI) {
        /* ......... */               /*Forced termination processing*/
    }

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

Note The RI600V4 do only simple error detection for *blk*. If *blk* is illegal and the error is not detected, the operation is not guaranteed after that.

7.3.5 Reference variable-sized memory pool state

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

- ref_mpl, iref_mpl
  These service calls store the detailed information (ID number of the task at the head of the wait queue, total size of free memory blocks, etc.) of the variable-size memory pool specified by parameter *mplid* into the area specified by parameter *pk_rmpl*.
  The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      mplid = 1;              /*Declares and initializes variable*/
    T_RMPL  pk_rmpl;                /*Declares data structure*/
    ID      wtskid;                 /*Declares variable*/
    SIZE    fmplsz;                 /*Declares variable*/
    UINT    fblksz;                 /*Declares variable*/

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

    ref_mpl (mplid, &pk_rmpl);      /*Reference variable-sized memory pool state*/

    wtskid = pk_rmpl.wtskid;        /*Reference ID number of the task at the */
                                    /*head of the wait queue*/
    fmplsz = pk_rmpl.fmplsz;        /*Reference total size of free memory blocks*/
    fblksz = pk_rmpl.fblksz;        /*Reference maximum memory block size*/

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

Note For details about the variable-sized memory pool state packet, refer to "[Variable-sized memory pool state packet: T_RMPL]".

CHAPTER 8 TIME MANAGEMENT FUNCTIONS

This chapter describes the time management functions performed by the RI600V4.

8.1 Outline

The RI600V4's time management function provides methods to implement time-related processing (Timer Operations: Delay task, Time-out, Cyclic handlers, Alarm Handlers and System Time) by using base clock timer interrupts that occur at constant intervals, as well as a function to manipulate and reference the system time.

8.2 System Time

The system time is a time used by the RI600V4 for performing time management (in millisecond).
After initialization to 0 by the Kernel Initialization Module (vsta_knl, ivsta_knl), the system time is updated based on the base clock interval defined by Denominator of base clock interval time (tic_deno) and Denominator of base clock interval time (tic_deno) in System Information (system) when creating a system configuration file.

8.2.1 Base clock timer interrupt

To realize the time management function, the RI600V4 uses interrupts that occur at constant intervals (base clock timer interrupts).
When a base clock timer interrupt occurs, processing related to the RI600V4 time (system time update, task time-out/ delay, cyclic handler activation, alarm handler activation, etc.) is executed.
Basically, either of channel 0-3 of the compare match timer (CMT) implemented in the MCU is used for base clock time. The channel number is specified by Selection of timer channel for base clock (timer)in Base Clock Interrupt Information (clock). in the system configuration file.
The hardware initialization to generate base clock timer interrupt is implemented by “void __RI_init_cmt(void)” in “ri_cmt.h”. The “ri_cmt.h” file is generated by the cfg600. The Boot processing function (PowerON_Reset_PC( )) must call _RI_init_cmt().

8.2.2 Base clock interval

In the RI600V4, service call parameters for time specification are specified in msec units.
It is desirable to set 1 msec for the occurrence interval of base clock timer interrupts, but it may be difficult depending on the target system performance (processing capability, required time resolution, or the like).
In such a case, the occurrence interval of base clock timer interrupt can be specified by Denominator of base clock interval time (tic_deno) and Denominator of base clock interval time (tic_deno) in System Information (system) when creating a system configuration file.
By specifying the base clock interval, processing regards that the time equivalent to the base clock interval elapses during a base clock timer interrupt.

8.3 Timer Operations

The RI600V4's timer operation function provides Delay task, Time-out, Cyclic handlers, Alarm Handlers and System Time, as the method for realizing time-dependent processing.

8.4 Delay task

Delayed task 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 wake-up is implemented by issuing the following service call from the processing program.

dly_tsk

8.5 Time-out

Time-out 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 time-out is implemented by issuing the following service call from the processing program.

tslp_tsk, twai_sem, twai_flg, tsnd_dtq, trcv_dtq, trcv_mbx, tloc_mtx, tsnd_mbf, trcv_mbf, tget_mpf, tget_mpl

8.6 Cyclic handlers

The cyclic handler is a routine dedicated to cycle processing that is activated periodically at a constant interval (activation cycle).
The RI600V4 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.

8.6.1 Basic form of cyclic handlers

The Extended information (exinf) in Cyclic Handler Information (cyclic_hand[]) is passed to the *exinf*.
The following shows the basic form of cyclic handlers.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

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

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

Note The cfg600 outputs the prototype declaration for the handler function to kernel_id.h.

8.6.2 Processing in cyclic handler

- Stack
  A cyclic handler uses the system stack.

- Service call
  The RI600V4 handles the cyclic handler as a “non-task”.
  The cyclic handler can issue service calls whose “Useful range” is “Non-task”.

  Note If a service call (isig_sem, iset_flg, etc.) which causes 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 RI600V4 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.

- PSW register when processing is started

Table 8-1 PSW Register When Cyclic Handler is Started

Bit	Value	Note
I	1	
IPL	Base clock interrupt priority level (IPL)	Do not lower IPL more than the start of processing.
PM	0	Supervisor mode
U	0	System stack
C, Z, S, O	Undefined	
Others	0	

8.6.3 Create cyclic handler

In the RI600V4, the method of creating a cyclic handler is limited to “static creation”.
Cyclic handlers therefore cannot be created dynamically using a method such as issuing a service call from a processing program.
Static cyclic handler creation means defining of cyclic handlers using static API “cyclic_hand[]” in the system configuration file.
For details about the static API “cyclic_hand[]”, refer to “19.16 Cyclic Handler Information (cyclic_hand[])”.

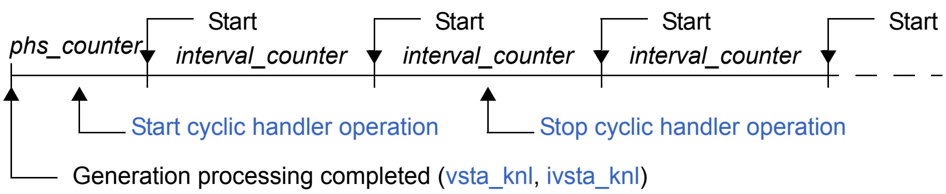
8.6.4 Start cyclic handler operation

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

- sta_cyc, ista_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 RI600V4.
  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 (phsatr) is specified for the target cyclic handler during configuration.

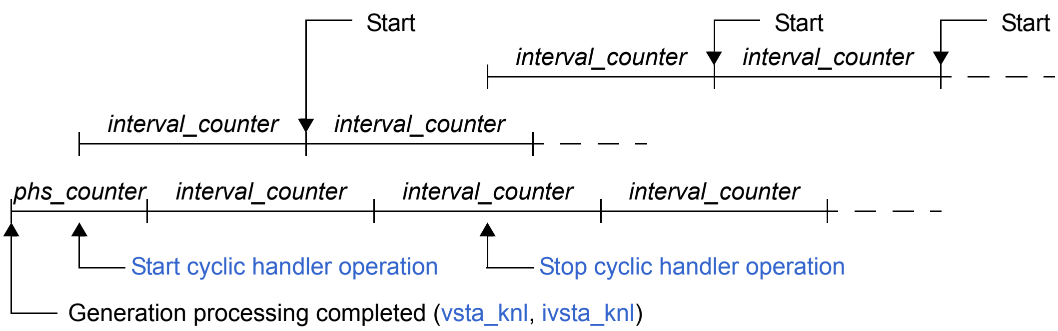
  - If the TA_PHS attribute is specified
    The target cyclic handler activation timing is set based on the Activation phase (phs_counter) and Activation cycle (interval_counter) defined during configuration.
    If the target cyclic handler has already been started, however, no processing is performed even if this service call is issued, but it is not handled as an error.
    The following shows a cyclic handler activation timing image.

Figure 8-1 TA_PHS Attribute: Specified



  - If the TA_PHS attribute is not specified
    The target cyclic handler activation timing is set based on the activation phase (Activation cycle (interval_counter)) 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.

Figure 8-2 TA_PHS Attribute: Not Specified



The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

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

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

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

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

8.6.5 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, istp_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 RI600V4 until issuance of sta_cyc or ista_cyc.
  The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      cycid = 1;              /*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.

8.6.6 Reference cyclic handler state

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

- ref_cyc, iref_cyc
  Stores cyclic handler state packet (current state, time until the next activation, etc.) of the cyclic handler specified by parameter *cycid* in the area specified by parameter *pk_rcyc*.
  The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      cycid = 1;              /*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 current state*/
    lefttim = pk_rcyc.lefttim;      /*Reference time left before the next */
                                    /*activation*/

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

Note For details about the cyclic handler state packet, refer to "[Cyclic handler state packet: T_RCYC]".

8.7 Alarm Handlers

The alarm handler is a routine started when the specified time passes.
The RI600V4 handles the alarm 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 time has elapsed, and the control is passed to the alarm handler.

8.7.1 Basic form of alarm handler

The Extended information (exinf) in Alarm Handler Information (alarm_handl[]) is passed to the *exinf*.
The following shows the basic form of alarm handlers.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

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

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

Note The cfg600 outputs the prototype declaration for the handler function to kernel_id.h.

8.7.2 Processing in alarm handler

- Stack
  A alarm handler uses the system stack.

- Service call
  The RI600V4 handles the alarm handler as a “non-task”.
  The alarm handler can issue service calls whose “Useful range” is “Non-task”.

  Note If a service call (isig_sem, iset_flg, etc.) which causes dispatch processing (task scheduling processing) is issued in order to quickly complete the processing in the alarm handler during the interval until the processing in the alarm handler ends, the RI600V4 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 alarm handler, upon which the actual dispatch processing is performed in batch.

- PSW register when processing is started

Table 8-2 PSW Register When Alarm Handler is Started)

Bit	Value	Note
I	1	
IPL	Base clock interrupt priority level (IPL)	Do not lower IPL more than the start of processing.
PM	0	Supervisor mode
U	0	System stack
C, Z, S, O	Undefined	
Others	0	

8.7.3 Create alarm handler

In the RI600V4, the method of creating a alarm handler is limited to “static creation”.
Alarm handlers therefore cannot be created dynamically using a method such as issuing a service call from a processing program.
Static alarm handler creation means defining of alarm handlers using static API “alarm_hand[]” in the system configuration file.
For details about the static API “alarm_hand[]”, refer to “19.17 Alarm Handler Information (alarm_handl[])”.

8.7.4 Start alarm handler operation

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

- sta_alm, ista_alm
  This service call sets the activation time of the alarm handler specified by *almid* in *almtim* (msec), and moves the alarm handler from the non-operational state (STP state) to operational state (STA state).
  As a result, the target alarm handler is handled as an activation target of the RI600V4.
  The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

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

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

    sta_alm (almid);                /*Start alarm handler operation*/

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

Note 1 When 0 is specified for *almtim*, the alarm handler will start at the next base clock interruption.

Note 2 When the target alarm handler has already started (STA state), this service call sets the activation time of the target alarm handler in *almtim* (msec) after canceling the activation time.

8.7.5 Stop alarm handler operation

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

- stp_alm, istp_alm
  This service call moves the alarm handler specified by parameter *cycid* from the operational state (STA state) to non-operational state (STP state).
  As a result, the target alarm handler is excluded from activation targets of the RI600V4 until issuance of sta_alm or ista_alm.
  The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

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

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

    stp_alm (almid);                /*Stop alarm handler operation*/

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

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

8.7.6 Reference alarm handler state

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

- ref_alm, iref_alm
  Stores alarm handler state packet (current state, time until the next activation, etc.) of the alarm handler specified by parameter *cycid* in the area specified by parameter *pk_rcyc*.
  The following describes an example for coding these service calls.

```
#include    "kernel.h"             /*Standard header file definition*/
#include    "kernel_id.h"          /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    ID      almid = 1;             /*Declares and initializes variable*/
    T_RALM  pk_ralm;               /*Declares data structure*/
    STAT    almstat;               /*Declares variable*/
    RELTIM  lefttim;               /*Declares variable*/

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

    ref_alm (almid, &pk_ralm);     /*Reference alarm handler state*/

    almstat = pk_ralm.almstat;     /*Reference current state*/
    lefttim = pk_ralm.lefttim;     /*Reference time left */

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

Note For details about the alarm handler state packet, refer to “[Alarm handler state packet: T_RALM]”.

8.8 System Time

8.8.1 Set system time

The system time can be set by issuing the following service call from the processing program.
Note that even if the system time is changed, the actual time at which the time management requests made before that (e.g., task time-outs, task delay by dly_tsk, cyclic handlers, and alarm handlers) are generated will not change.

- set_tim, iset_tim
  These service calls change the system time (unit: msec) to the time specified by parameter *p_systim*.
  The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    SYSTIM  p_systim;               /*Declares data structure*/

    p_systim.ltime = 3600;          /*Initializes data structure*/
    p_systim.utime = 0;             /*Initializes data structure*/

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

    set_tim (&p_systim);            /*Set system time*/

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

Note For details about the system time packet SYSTIM, refer to "[System time packet: SYSTIM]".

8.8.2 Reference system time

The system time can be referenced by issuing the following service call from the processing program.

- get_tim, iget_tim
  These service calls store the system time (unit: msec) into the area specified by parameter *p_systim*.
  The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    SYSTIM  p_systim;               /*Declares data structure*/
    UW      ltime;                  /*Declares variable*/
    UH      utime;                  /*Declares variable*/

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

    get_tim (&p_systim);            /*Reference System Time*/

    ltime = p_systim.ltime;         /*Acquirer system time (lower 32 bits)*/
    utime = p_systim.utime;         /*Acquirer system time (higher 16 bits)*/

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

Note For details about the system time packet SYSTIM, refer to “[System time packet: SYSTIM]”.

8.9 Initialize Base Clock Timer

The cfg600 outputs the file “ri_cmt.h” which the base clock timer initialization function (void _RI_init_cmt(void)) is described. The Boot processing function (PowerON_Reset_PC( )) should call the base clock timer initialization function.

CHAPTER 9 SYSTEM STATE MANAGEMENT FUNCTIONS

This chapter describes the system management functions performed by the RI600V4.

9.1 Outline

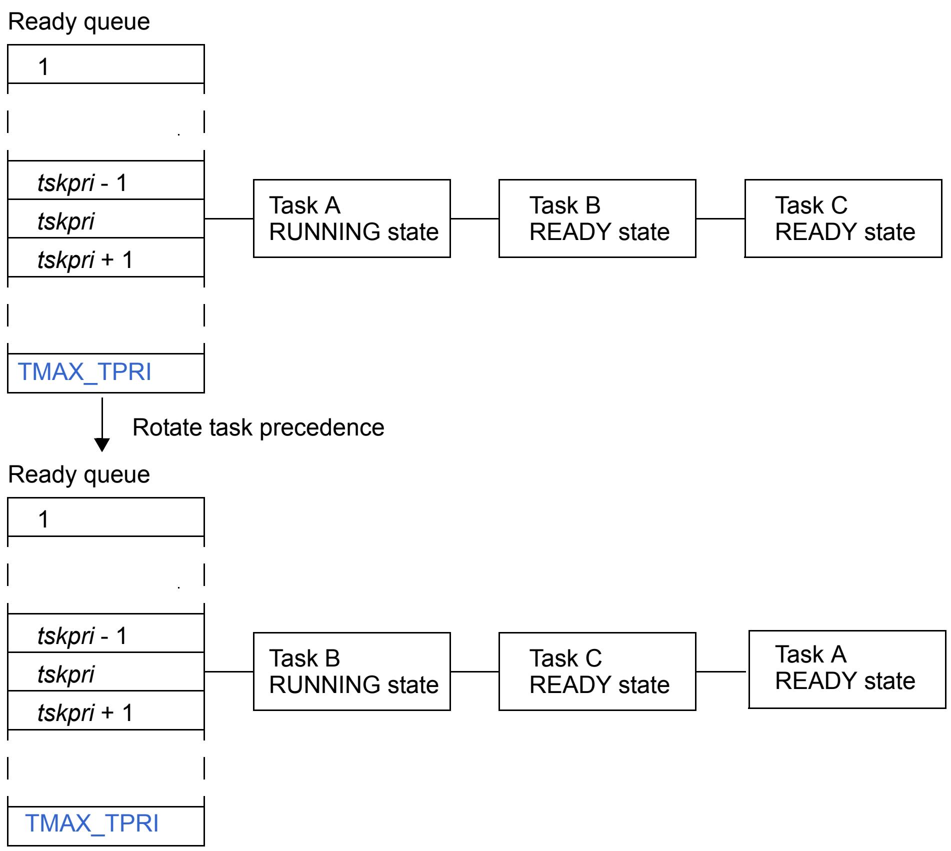
The RI600V4's system status management function provides functions for referencing the system status such as the context type and CPU lock status, as well as functions for manipulating the system status such as ready queue rotation, scheduler activation, or the like.
Note, refer to "CHAPTER 13 SYSTEM DOWN" for system down (vsys_dwn, ivsys_dwn) and refer to "CHAPTER 16 SYSTEM INITIALIZATION" for starting of the RI600V4 (vsta_knl, ivsta_knl).

9.2 Rotate Task Precedence

Task precedence 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 9-1 Rotate Task Precedence



The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void cychdr (VP_INT exinf)          /*Cyclic handler*/
{
    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 RI600V4'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.

Note 4 When TPRI_SELF is specified as *tskpri*, the base priority of the invoking task is applied as the target priority of this service call.
As for a task which has locked mutexes, the current priority might be different from the base priority. In this case, even if the task issues this servie call specifying TPRI_SELF as parameter *tskpri*, the ready queue of the current priority that the invoking task belongs cannot be changed.

9.3 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 these service calls.

```
#include    "kernel.h"               /*Standard header file definition*/
#include    "kernel_id.h"            /*Header file generated by cfg600*/

void inthdr (void)                   /*Interrupt handler*/
{
    ID      p_tskid;                 /*Declares variable*/

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

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

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

    return;                          /*Terminate interrupt 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.

9.4 Lock and Unlock the CPU

In the CPU locked state, the task scheduling is prohibited, and kernel interrupts are masked. Therefore, exclusive processing can be achieved for all processing programs except non-kernel interrupt handlers.
The following service calls moves to the CPU locked state.

- loc_cpu, iloc_cpu
  These service calls transit the system to the CPU locked state.
  The service calls that can be issued in the CPU locked state are limited to the one listed below.

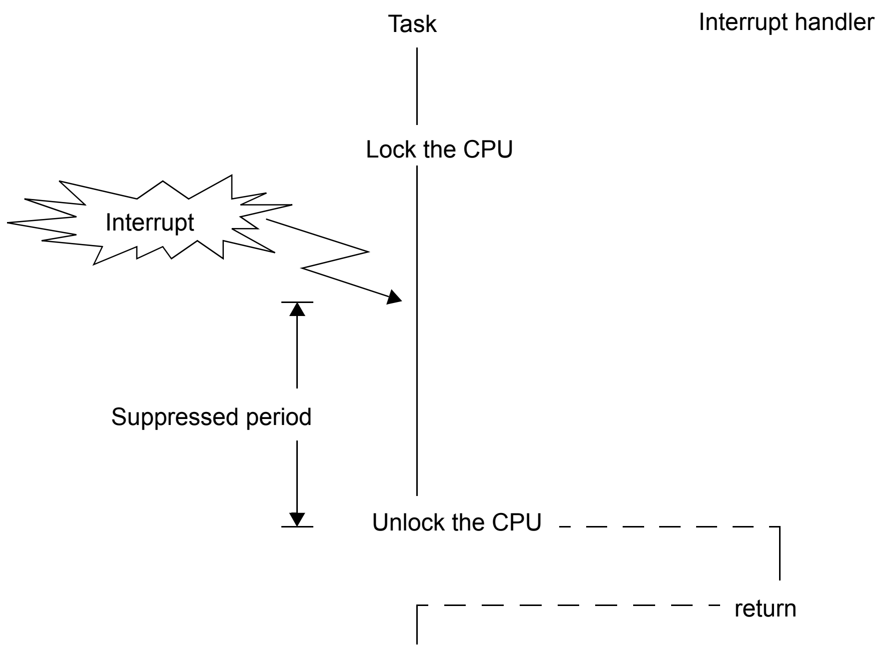
Service Call that can be issued	Function
ext_tsk	Terminate invoking task. (This service call transit the system to the CPU unlocked state.)
loc_cpu, iloc_cpu	Lock the CPU.
unl_cpu, iunl_cpu	Unlock the CPU.
sns_loc	Reference CPU state.
sns_dsp	Reference dispatching state.
sns_ctx	Reference contexts.
sns_dpn	Reference dispatch pending state.
vsys_dwn, ivsys_dwn	System down

The following service calls and ext_tsk release from the CPU locked state.

- unl_cpu, iunl_cpu
  These service calls transit the system to the CPU unlocked state.

The following shows a processing flow when using the CPU locked state.

Figure 9-2 Lock the CPU



The following describes an example for coding “lock the CPU” and “unlock the CPU”.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void 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 loc_cpu or iloc_cpu must be cancelled before the processing program that issued this service call ends.

Note 2 The loc_cpu and iloc_cpu do 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 unl_cpu and iunl_cpu do not perform queuing of unlock requests. If the system is in the CPU unlocked state, therefore, no processing is performed but it is not handled as an error

Note 4 The unl_cpu and iunl_cpu do not cancel the dispatching disabled state that was set by issuing dis_dsp.

Note 5 The base clock interrupt is masked during the CPU locked state. Therefore, time handled by the TIME MANAGEMENT FUNCTIONS may be delayed if the period of the CPU locked state becomes long.

Note 6 For kernel interrupts, refer to "10.1 Interrupt Type".

9.5 Reference CPU Locked State

It may be necessary to refer to current CPU locked state in functions that are called from two or more tasks and handlers. In this case, sns_loc is useful.

- sns_loc
  This service call examines whether the system is in the CPU locked state or not. This service call returns TRUE when the system is in the CPU locked state, and return FALSE when the system is in the CPU unlocked state.
  The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

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

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

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

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

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

9.6 Disable and Enable Dispatching

In the dispatching disabled state, the task scheduling is prohibited. Therefore, exclusive processing can be achieved for all tasks.
The following service call moves to the dispatching disabled state. And also when PSW.IPL is changed to other than 0 by using chg_ims, the system shifts to the dispatching disabled state.

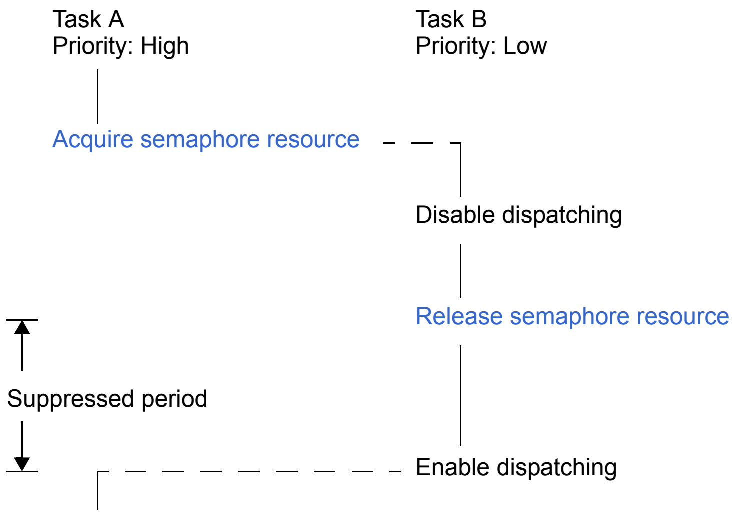
- dis_dsp
  This service call transits the system to the dispatching disabled state.

The dispatching disabled state is cancelled by the following service call, ext_tsk, and chg_ims that changes PSW.IPL to 0.

- ena_dsp
  This service call transits the system to the dispatching enabled state.

The following shows a processing flow when using the dispatching disabled state.

Figure 9-3 Disable Dispatching



The following describes an example for coding this service call.

```
#include    "kernel.h"            /*Standard header file definition*/
#include    "kernel_id.h"         /*Header file generated by cfg600*/

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

    dis_dsp ();                   /*Disable dispatching*/

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

    ena_dsp ();                   /*Enable dispatching*/

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

Note 1 The dispatching disabled state must be cancelled before the task that issued dis_dsp moves to the DORMANT state.

Note 2 The dis_dsp does not perform queuing of lock requests. If the system is in the dispatching disabled state, therefore, no processing is performed but it is not handled as an error.

Note 3 The ena_dsp does not perform queuing of unlock requests. If the system is in the dispatching enabled state, therefore, no processing is performed but it is not handled as an error

Note 4 If a service call (such as wai_sem, wai_flg) that may move the status of the invoking task is issued while the dispatching disabled state, that service call returns E_CTX regardless of whether the required condition is immediately satisfied.

9.7 Reference Dispatching State

It may be necessary to refer to current dispatching disabled state in functions that are called from two or more tasks . In this case, sns_dsp is useful.

- sns_dsp
  This service call examines whether the system is in the dispatching disabled state or not. This service call returns TRUE when the system is in the dispatching disabled state, and return FALSE when the system is in the dispatching enabled state.
  The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

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

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

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

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

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

9.8 Reference Context Type

It may be necessary to refer to current context type in functions that are called from two or more tasks and handlers. In this case, sns_ctx is useful.

- sns_ctx
  This service call examines the context type of the processing program that issues this service call. This service call returns TRUE when the processing program is non-task context, and return FALSE when the processing program is task context.
  The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

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

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

    ercd = sns_ctx ( );             /*Reference context type*/

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

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

9.9 Reference Dispatch Pending State

The state to fill either the following is called dispatch pending state.

- Dispatching disabled state
- CPU locked state
- PSW.IPL > 0, such as handlers

It may be necessary to refer to current dispatch pending state in functions that are called from two or more tasks and handlers. In this case, sns_dpn is useful.

- sns_dpn
  This service call examines whether the system is in the dispatch pending state or not. This service call returns TRUE when the system is in the dispatch pending state, and return FALSE when the system is not in the dispatch pending state.
  The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

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

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

    ercd = sns_dpn ();              /*Reference dispatch pending state*/

    if (ercd == TRUE) {
        /* ......... */             /*Dispatch pending state*/
    } else if (ercd == FALSE) {
        /* ......... */             /*Other state*/
    }

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

CHAPTER 10 INTERRUPT MANAGEMENT FUNCTIONS

This chapter describes the interrupt management functions performed by the RI600V4.

10.1 Interrupt Type

Interrupts are classified into kernel interrupt and non-kernel interrupt.

- Kernel interrupt
  An interrupt whose interrupt priority level is lower than or equal to the kernel interrupt mask level is called the kernel interrupt.
  A kernel interrupt handler can issue service calls.
  Note, however, that handling of kernel interrupts generated during kernel processing may be delayed until the interrupts become acceptable.

- Non-kernel interrupt
  An interrupt whose interrupt priority level is higher than the kernel interrupt mask level is called the non-kernel interrupt. The non-maskable interrupt is classified into non-kernel interrupt.
  A non-kernel interrupt handler must not issue service calls.
  Non-kernel interrupts generated during service-call processing are immediately accepted whether or not kernel processing is in progress.

Note The kernel interrupt mask level id defined by Kernel interrupt mask level (system_IPL) in System Information (system).

10.2 Fast Interrupt of the RX-MCU

The RX-MCU supports the “fast interrupt” function. Only one interrupt source can be made the fast interrupt. The fast interrupt is handled as the one that has interrupt priority level 15. To use the fast interrupt function, make sure there is only one interrupt source that is assigned interrupt priority level 15.
For the fast interrupt function to be used in the RI600V4, it is necessary that the interrupt concerned be handled as an non-kernel interrupt. In other words, the kernel interrupt mask level must be set to 14 or below.
And “os_int = NO;” and “pragma_switch = F;” are required for interrupt_vector[] definition.
And the FINTV register of the RX-MCU must be initialized to the start address of the handler in the Boot processing function (PowerON_Reset_PC( )).

10.3 CPU Exception

The following CPU exceptions are handled as non-kernel interrupt.

- Unconditional trap (INT, BRK instruction)
  Note, INT #1 to #8 are reserved by the RI600V4.
- Undefined instruction exception
- Privileged instruction exception
- Floating-point exception

On the other hand, the access exception is handled as kernel interrupt.

10.4 Base Clock Timer Interrupt

The TIME MANAGEMENT FUNCTIONS is realized by using base clock timer interrupts that occur at constant intervals. When the base clock timer interrupt occurs, The RI600V4's time management interrupt handler is activated and executes time-related processing (system time update, delayed wake-up/time-out of task, cyclic handler activation, etc.).

10.5 Multiple Interrupts

In the RI600V4, occurrence of an interrupt in an interrupt handler is called “multiple interrupts”.
It can be set whether each interrupt handler for relocatable vector permits multiple interrupts. For details, refer to “19.18 Relocatable Vector Information (interrupt_vector[])”.

10.6 Interrupt Handlers

The interrupt handler is a routine dedicated to interrupt servicing that is activated when an interrupt occurs.
The RI600V4 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.

10.6.1 Basic form of interrupt handlers

The following shows the basic form of interrupt handlers.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void inthdr (void)
{
    /* ......... */

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

Note The cfg600 outputs the prototype declaration and #pragma interrupt directive for the handler function to kernel_id.h.

- Stack
  A interrupt handler uses the system stack.

- Service call
  The RI600V4 handles the interrupt handler as a “non-task”.
  The kernel interrupt handler can issue service calls whose “Useful range” is “Non-task”.
  No service call can be issued in non-kernel interrupt handler.

- If a service call (isig_sem, iset_flg, etc.) which causes 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 RI600V4 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.

- PSW register when processing is started

Table 10-1 PSW Register When Interrupt Handler is Started

Bit	Value	Note
I	- “pragma_switch = E”: 1 - Other cases: 0	
IPL	- Interrupt: Interrupt priority level - CPU exception: Same before exception	Do not lower IPL more than the start of processing.
PM	0	Supervisor mode
U	0	System stack
C, Z, S, O	Undefined	
Others	0	

10.6.2 Register interrupt handler

The RI600V4 supports the static registration of interrupt handlers only. They cannot be registered dynamically by issuing a service call from the processing program.
Static interrupt handler registration means defining of interrupt handlers using static API “interrupt_vector[]” (relocatable vector) and “interrupt_fvector[]” (fixed vector/exception vector) in the system configuration file.
For details about the static API “interrupt_vector[]”, refer to “19.18 Relocatable Vector Information (interrupt_vector[])”, and for details about the static API “interrupt_fvector[]”, refer to “19.19 Fixed Vector/Exception Vector Information (interrupt_fvector[])”.

10.7 Maskable Interrupt Acknowledgement Status in Processing Programs

The maskable interrupt acknowledgement status of RX-MCU depends on the values of PSW.I and PSW.IPL. See the hardware manual for details.
The initial status is determined separately for each processing program. See Table 10-2 for details.

Table 10-2 Maskable Interrupt Acknowledgement Status upon Processing Program Startup

Processing Program	PSW.I	PSW.IPL
Task	1	0
Cyclic handler, Alarm handler	1	Base clock interrupt priority level (IPL)
Interrupt Handler	- “pragma_switch = E”: 1 - Other cases: 0	- Interrupt: Interrupt priority level - CPU exception: Same before exception

10.8 Prohibit Maskable Interrupts

There is the following as a method of prohibiting maskable interrupts.

- Move to the CPU locked state by using loc_cpu, iloc_cpu
- Change PSW.IPL by using chg_ims, ichg_ims
- Change PSW.I and PSW.IPL directly (only for handlers)

10.8.1 Move to the CPU locked state by using loc_cpu, iloc_cpu

In the CPU locked state, PSW.IPL is changed to the Kernel interrupt mask level (system_IPL). Therefore, only kernel interrupts are prohibited in the CPU locked state.
Note, in the CPU locked state, service call issuance is restricted. For details, refer to "9.4 Lock and Unlock the CPU".

10.8.2 Change PSW.IPL by using chg_ims, ichg_ims

The PSW.IPL can be changed to arbitrary value by using chg_ims, ichg_ims.
When a task changes PSW.IPL to other than 0 by using chg_ims, the system is moved to the dispatching disabled state.
When a task returns PSW.IPL to 0, the system returns to the dispatching enabled state.
Do not issue ena_dsp while a task changes PSW.IPL to other than 0 by using chg_ims. If issuing ena_dsp, the system moves to the dispatching enabled state. If task dispatching occurs, PSW is changed for the dispatched task. Therefore PSW.IPL may be lowered without intending it.
The handlers must not lower PSW.IPL more than it starts.

10.8.3 Change PSW.I and PSW.IPL directly (only for handlers)

The handlers can change PSW.I and PSW.IPL directly. This method is faster than ichg_ims.
The handlers must not lower PSW.IPL more than it starts.
Note, the compiler provides following intrinsic functions for operating PSW. See CubeSuite+ RX Build User's Manual for details about intrinsic functions.

- set_ipl(): Change PSW.IPL
- get_ipl(): Refer to PSW.IPL
- set_psw(): Change PSW
- get_psw(): Refer to PSW

CHAPTER 11 SYSTEM CONFIGURATION MANAGEMENT FUNCTIONS

This chapter describes the system configuration management functions performed by the RI600V4.

11.1 Outline

The RI600V4's system configuration management function provides the function to reference the version information.

11.2 Reference Version Information

The version information can be referenced by issuing the following service call from the processing program.

- ref_ver, iref_ver
  These service calls store the version information into the area specified by parameter *pk_rver*.
  The following describes an example for coding these service calls.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void task (VP_INT exinf)
{
    T_RVER  pk_rver;                /*Declares data structure*/
    UH      maker;                  /*Declares variable*/
    UH      prid;                   /*Declares variable*/

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

    ref_ver (&pk_rver);             /*Reference version information/

    maker = pk_rver.maker;          /*Acquirer system time (lower 32 bits)*/
    prid = pk_rver.prid;            /*Acquirer system time (higher 16 bits)*/

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

Note For details about the version information packet T_RVER, refer to “[Version information packet: T_RVER]”.

CHAPTER 12 OBJECT RESET FUNCTIONS

This chapter describes the object reset functions performed by the RI600V4.

12.1 Outline

The object reset function returns Data Queues, Mailboxes, Message Buffers, Fixed-Sized Memory Pools and Variable-Sized Memory Pools to the initial state. The object reset function falls outside μITRON4.0 specification.

12.2 Reset Data Queue

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

- vrst_dtq
  This service call reset the data queue specified by parameter *dtqid*.
  The data having been accumulated by the data queue area are annulled. The tasks to wait to send data to the target data queue are released from the WAITING state, and EV_RST is returned as a return value for the tasks.
  The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

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

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

    ercd = vrst_dtq ( dtqid );  /*Reset data queue*/

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

Note In this service call, the tasks to wait to receive data do not released from the WAITING state.

12.3 Reset Mailbox

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

- vrst_mbx
  This service call reset the mailbox specified by parameter *mbxid*.
  The messages having been accumulated by the mailbox come off from the management of the RI600V4.
  The following describes an example for coding this service call.

```
#include    "kernel.h"               /*Standard header file definition*/
#include    "kernel_id.h"            /*Header file generated by cfg600*/

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

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

    ercd = vrst_mbx ( mbxid) ;   /*Reset mailbox*/

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

Note In this service call, the tasks to wait to receive message do not released from the WAITING state.

12.4 Reset Message Buffer

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

- vrst_mbf
  This service call reset the message buffer specified by parameter *mbfid*.
  The messages having been accumulated by the message buffer area are annulled. The tasks to wait to send message to the target message buffer are released from the WAITING state, and EV_RST is returned as a return value for the tasks.
  The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

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

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

    ercd = vrst_mbf ( mbfid );  /*Reset message buffer*/

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

Note In this service call, the tasks to wait to receive message do not released from the WAITING state.

12.5 Reset Fixed-sized Memory Pool

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

- vrst_mpf
  This service call reset the fixed-sized memory pool specified by parameter *mpfid*.
  The tasks to wait to get memory block from the target fixed-sized memory pool are released from the WAITING state, and EV_RST is returned as a return value for the tasks.
  All fixed-sized memory blocks that had already been acquired are returned to the target fixed-sized memory pool. Therefore, do not access those fixed-sized memory blocks after issuing this service call.
  The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

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

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

    ercd = vrst_mpf ( mpfid );   /*Reset fixed-sized memory pool*/

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

12.6 Reset Variable-sized Memory Pool

A variable-sized memory pool is reset by issuing the following service call from the processing program.

- vrst_mpl
  This service call reset the variable-sized memory pool specified by parameter *mpfid*. The tasks to wait to get memory block from the target variable-sized memory pool are released from the WAITING state, and EV_RST is returned as a return value for the tasks.
  All variable-sized memory blocks that had already been acquired are returned to the target variable-sized memory pool. Therefore, do not access those variable-sized memory blocks after issuing this service call.
  The following describes an example for coding this service call.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

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

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

    ercd = vrst_mpl ( mplid );  /*Reset variable-sized memory pool*/

    if (ercd == E_OK) {

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

Note All variable-sized memory blocks that had already been acquired are returned to the target variable-sized memory pool. Therefore, do not access those variable-sized memory blocks after issuing this service call.

CHAPTER 13 SYSTEM DOWN

This chapter describes the system down functions performed by the RI600V4.

13.1 Outline

When the event that cannot be recovered while the RI600V4 is operating occurs, the system down is caused and the system down routine is invoked.

13.2 User-Own Coding Module

The system down routine must be implemented as user-own coding module.

Note The System down routine (_RI_sys_dwn__) which is provided by the RI600V4 as a sample file is implemented in the boot processing file “resetprg.c”.

13.2.1 System down routine (_RI_sys_dwn__)

The following shows the basic form of the system down routine. The system down routine must not return.

```
#include    "kernel.h"              /*Standard header file definition*/
#include    "kernel_id.h"           /*Header file generated by cfg600*/

void  _RI_sys_dwn__ ( W type, VW inf1, VW inf2, VW inf3 ); /*Prototype declaration*/

void  _RI_sys_dwn__ ( W type, VW inf1, VW inf2, VW inf3 )
{
    /* ......... */

        while(1);
}
```

Note The function name of the system down routine is “_RI_sys_dwn__”.

- Stack
  The system down routine uses the system stack.

- Service call
  The system down routine must not issue service calls.

- PSW register when processing is started

Table 13-1 PSW Register When System Down Routine is Started

Bit	Value	Note
I	0	
IPL	- *type* < 0 : Undefined - *type* >= 0 : Same before system down	Do not lower IPL more than the start of processing.
PM	0	Supervisor mode
U	0	System stack
C, Z, S, O	Undefined	
Others	0	

13.2.2 Parameters of system down routine

- *type* == -1 (Error when a kernel interrupt handler ends)

Table 13-2 Parameters of System Down Routine (*type* == -1)

inf1	inf2	inf3	Description
E_CTX (-25)	2	Undefined	PSW.PM == 1 (user mode) when a kernel interrupt handler ends.
	3	Undefined	PSW.IPL > kernel interrupt mask level when a kernel interrupt handler ends.
	5	Undefined	The system is in the CPU locked state when a kernel interrupt handler ends.

- *type* == -2 (Error in ext_tsk)

Table 13-3 Parameters of System Down Routine (*type* == -2)

inf1	inf2	inf3	Description
E_CTX (-25)	1	Undefined	The ext_tsk is called in the non-task context.
	4	Undefined	PSW.IPL > kernel interrupt mask level when ext_tsk is called.

- *type* == -3 (Unlinked service call issued)

Table 13-4 Parameters of System Down Routine (*type* == -3)

inf1	inf2	inf3	Description
E_NOSPT (-9)	Undefined	Undefined	Unlinked service call is issued.

Note Refer to "2.6.1 Service call information files and "-ri600_preinit_mrc" compiler option".

- *type* == -16 (Undefined relocatable vector interrupt)

Table 13-5 Parameters of System Down Routine (*type* == -16)

inf1	inf2	inf3
- "-U" option is not specified for cfg600 Undefined - "-U" option is specified for cfg600 Vector number	PC, which is pushed to the stack by CPU's interrupt operation	PSW, which is pushed to the stack by CPU's interrupt operation

- *type* == -17 (Undefined fixed vector/exception vector interrupt)

Table 13-6 Parameters of System Down Routine (type == -17)

inf1	inf2	inf3
- "-U" option is not specified for cfg600 Undefined - "-U" option is specified for cfg600 Vector number	PC, which is pushed to the stack by CPU's interrupt operation	PSW, which is pushed to the stack by CPU's interrupt operation

- *type* > 0 (Issuing vsys_dwn, ivsys_dwn from application))
  0 and a negative *type* value is reserved by the RI600V4. When calling vsys_dwn, ivsys_dwn from application, use positive *type* value.

Table 13-7 Parameters of System Down Routine (type > 0)

inf1	inf2	inf3
Value specified for vsys_dwn, ivsys_dwn		

CHAPTER 14 SCHEDULING FUNCTION

This chapter describes the scheduler of the RI600V4.

14.1 Outline

The scheduling functions provided by the RI600V4 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.

14.2 Processing Unit and Precedence

An application program is executed in the following processing units.

- Task
- Interrupt handler
- Cyclic handler
- Alarm handler

The various processing units are processed in the following order of precedence.

1) Interrupt handlers, cyclic handlers, alarm handlers

2) Scheduler

3) Tasks

The “scheduler” is the RI600V4’s processing that schedules running task and dispatches to the task.
Since interrupt handler, cyclic handlers and alarm handlers have higher precedence than the scheduler, no tasks are executed while these handlers are executing. (Refer to “14.7 Task Scheduling in Non-Tasks”).
The precedence of an interrupt handler becomes higher when the interrupt level is higher.
The precedence of a cyclic handler and alarm handler is the same as the interrupt handler which interrupt level is same as the base clock timer interrupt.
The order of precedence for tasks depends on the current priority of the tasks.

14.3 Task Drive Method

The RI600V4 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 RI600V4, 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 base clock interrupt used when achieving TIME MANAGEMENT FUNCTIONS

14.4 Task Scheduling Method

As task scheduling methods, the RI600V4 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 target to RI600V4 scheduling.

- Priority level method

  A task with the highest current priority is selected from among all the tasks that have entered an executable state (RUNNING state or READY state), and given the CPU use right.

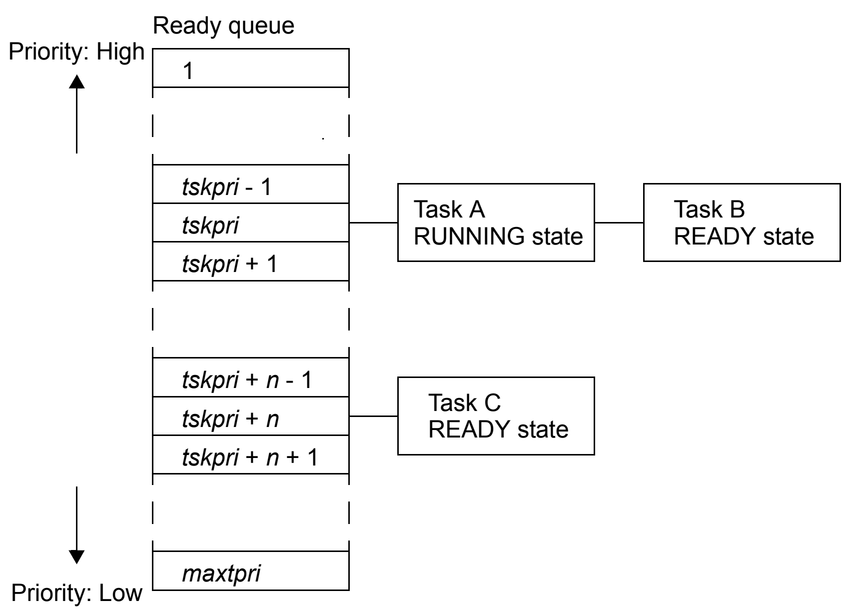
- FCFS method

  When two or more “task with the highest priority level” exist, the scheduling target task can not be decided only by the Priority level method. In this case, the RI600V4 decides the scheduling target task by first come first served (FCFS) method. Concretely, the task that enters to executable state (READY state) earliest among them, and given the CPU use right.

14.4.1 Ready queue

The RI600V4 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 RI600V4'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 14-1 Implementation of Scheduling Method (Priority Level Method or FCFS Method)



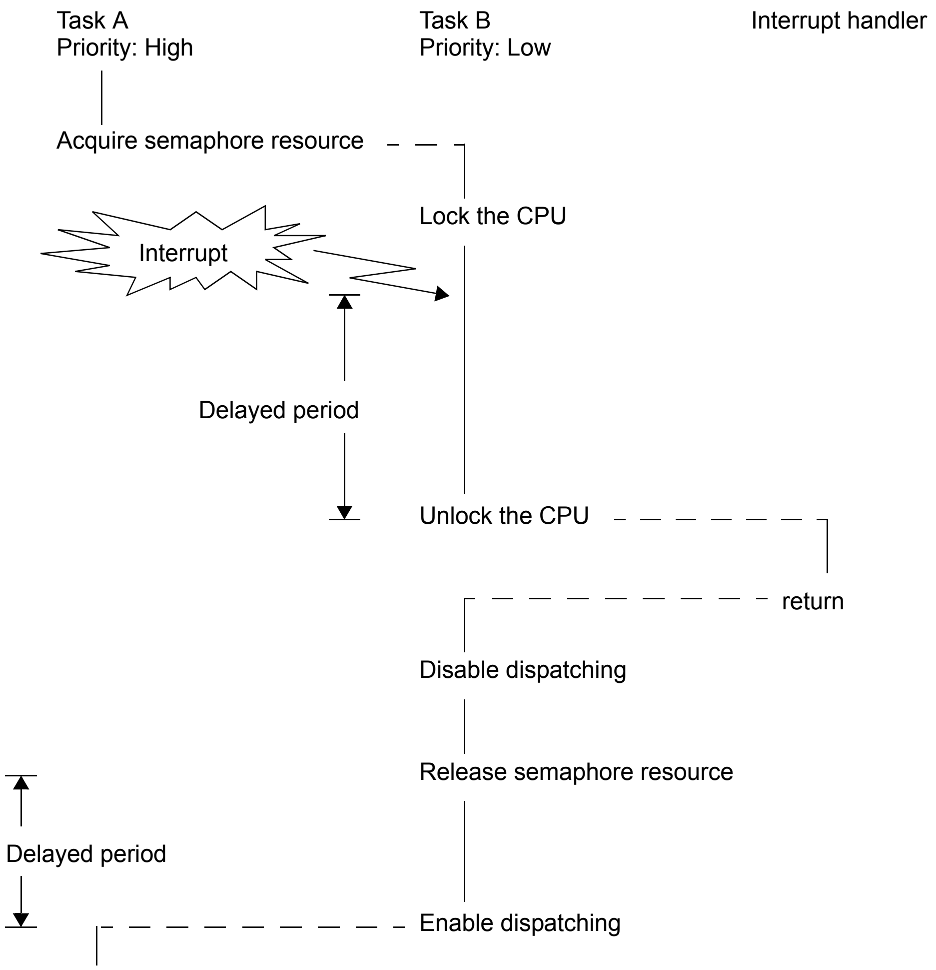
- Create ready queue

  In the RI600V4, the method of creating a ready queue is limited to “static creation”.
  Ready queues therefore cannot be created dynamically using a method such as issuing a service call from a processing program.
  Static ready queue creation means defining of Maximum task priority (priority) in System Information (system) in the system configuration file.

14.5 Task Scheduling Lock Function

The RI600V4 provides the scheduling lock function for manipulating the scheduler status explicitly from the processing program and disabling/enabling dispatch processing.
The following shows a processing flow when using the scheduling lock function.

Figure 14-2 Scheduling Lock Function



For details, refer to “9.4 Lock and Unlock the CPU” and “9.6 Disable and Enable Dispatching”.

14.6 Idling

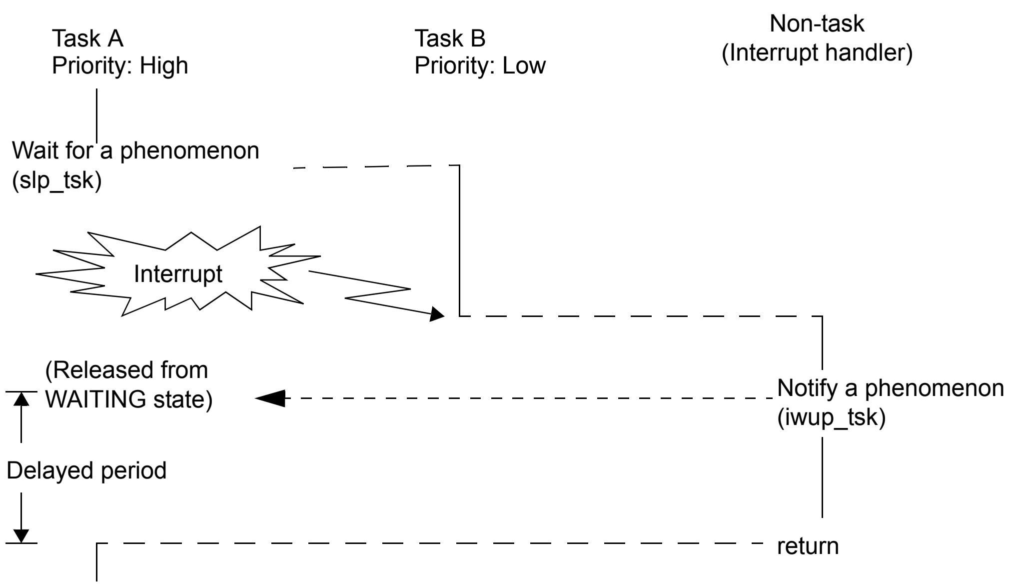
When there is no RUNNING or READY task, the RI600V4 enters an endless loop and waits for interrupts.

14.7 Task Scheduling in Non-Tasks

If processing of non-tasks starts, any tasks will not be performed until non-task processing is completed, since the precedence of non-task (interrupt handler, cyclic handler and alarm handler) is higher than task as shown in "14.2 Processing Unit and Precedence".

The following shows a example when a service call accompanying dispatch processing is issued in non-tasks. In this example, when the interrupt handler issues iwup_tsk, the Task A whose priority is higher than the task B is released from the WAITING state, but processing of the interrupt handler is continued at this time, without performing the task A yet. When processing of the interrupt handler is completed, the scheduler is started, and as a result, the task A is performed.

Figure 14-3 Scheduling in Non-Tasks



CHAPTER 15 REALTIME OS TASK ANALYZER

15.1 Outline

The following information is required when analyzing the system incorporating Realtime OS.

- The execution situation of processing programs
- The use situation of Realtime OS resources
- The CPU usage rate for every processing program

The tool for realizing the above is “Realtime OS Task Analyzer”. The Realtime OS Task Analyzer analyzes the information outputted by Realtime OS and displays it graphically.

This chapter describes the procedure for using Realtime OS Task Analyzer. See “RI600V4 Real-Time Operating System User's Manual: Analysis” for the functions and operation method of the Realtime OS Task Analyzer.

15.2 Trace Mode

There is the type of usage shown below in the Realtime OS Task Analyzer. The trace mode is selected in [Task Analyzer] tab.

- Taking in trace chart by hardware trace mode
  In this mode, the trace information is collected in the trace memory which emulator or simulator has.

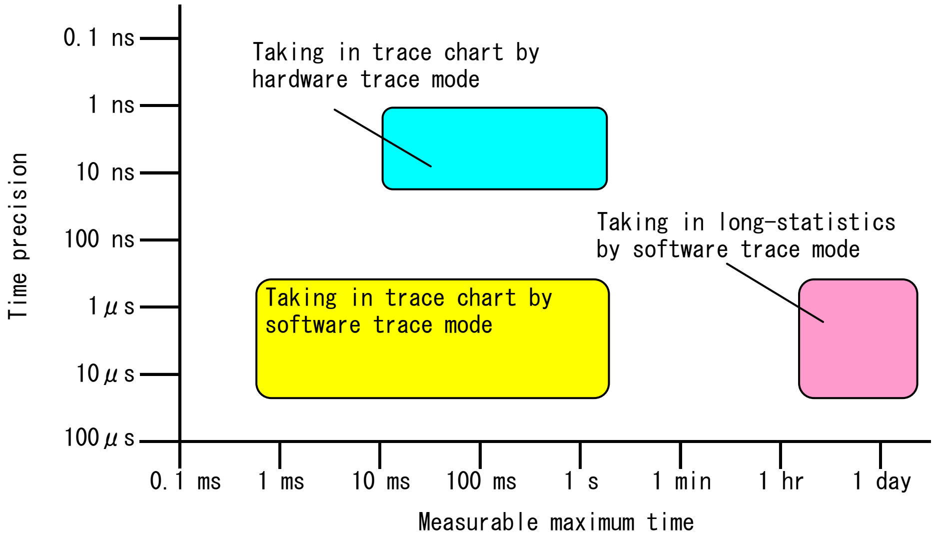
- Taking in trace chart by software trace mode
  In this mode, the trace information is collected in the trace buffer secured on the user memory area. The buffer size is specified in [Task Analyzer] tab. Please refer to “15.4 Trace Buffer Size (Taking in Trace Chart by Software Trace Mode)” for the estimate of the size of the trace buffer.
  To use this mode, implementation of user-own coding module and setup of the system configuration file are required. For details, refer to “15.3.1 Taking in trace chart by software trace mode”.

- Taking in long-statistics by software trace mode
  In this mode, the trace information is collected in the RI600V4’s variable secured on the user memory area. The size of this variable is roughly 2 K-bytes. For details, refer to “19.20.1 BRI_RAM section”.
  To use this mode, implementation of user-own coding module and setup of the system configuration file are required. For details, refer to “15.3.2 Taking in long-statistics by software trace mode”.

- Not tracing
  The Realtime OS Task Analyzer can not be used.

The measurable maximum time and the time precision differ for every trace mode. The standard is shown to Figure 15-1.

Figure 15-1 Measurable maximum time and the time precision



Note 1 In the "Taking in trace chart by hardware trace mode", the measurable maximum time depends on the size of the trace memory which the emulator or simulator has. And the time precision depends on the emulator or simulator specification.

Note 2 In the "Taking in trace chart by softwarre trace mode", the measurable maximum time depends on the size of the trace buffer. And refer to "15.3.1 Taking in trace chart by software trace mode" for the time precision.

Note 3 In the "Taking in long-statistics by software trace mode", refer to "15.3.2 Taking in long-statistics by software trace mode" for the measurable maximum time and the time precision.

When using the Realtime OS Task Analyzer, compared with the case where it is not used,it has the influence shown in Table 15-1 on the target system. Note, the processing time in Table 15-1 is approximate value when the CPU clock is 100MHz.

Table 15-1 Influence on Target System

	Taking in trace chart by hardware trace mode	Taking in trace chart by software trace mode	Taking in long-statistics by software trace mode
Service call processing time	Worse for about 0.5 - 1.5 μs (It depends on the number of tasks state change.)	Worse for about 1.5 - 5 μs (It depends on the number of tasks state change.)	No degradation
Task-dispatching processing time	Worse for about 0.2 μs	Worse for about 0.7 μs	Worse for about 0.6 μs
Interrupt processing time	Worse for about 0.5 μs	Worse for about 1 - 2 μs	Worse for about 1 - 2 μs
Consumption of RAM	No degradation	Needs a buffer	Roughly 2 K-bytes
Implementation of user-own coding module and setup of the system configuration file	Not required	Required	Required

Reference to the function of each mode , Figure 15-1 and Table 15-1, please decide the trace mode to be used.

The trace mode is selected in [Task Analyzer] tab. Then by performing a build, the load module which contains the Realtime OS module that corresponds the trace mode to be selected is generated.

15.3 User-Own Coding Module for Software Trace Mode

15.3.1 Taking in trace chart by software trace mode

In this mode, the RI600V4 gets time-stamp from user-own coding module. Usually, the hardware timer is used in order to generate time-stamp. The bit width of the counter of the hardware timer has necessity of 16 bits or more. Note, CMT (Compare Match Timer), which is built in RX family MCU as standard, satisfies this requirement.
This section describes the specification of function and variables to be implemented as user-own coding module. Since each function does not follow ABI (Application Binary Interface) of the RX family C/C++ compiler, it needs to be implemented by using assembly language. In this section, function and variable name are described in assembly language level.

Note The sample file provided by the RI600V4 is “trcSW_cmt.src”. This file uses CMT channel-1.

1) __RIUSR_trcSW_base_time (Time precision)

Define the unit of the time returned by __RIUSR_trcSW_read_cnt (Function to get time-stamp) as a constant for the 32 bit- unsigned integer. Usually, please set up the time of 1 count of hardware timer counter.
A typical setup in the case of using CMT is shown below.

PCLK	Dividing rate	Time precision (see Note)
12.5 MHz	8	0.64 μs
	32	2.56 μs
	128	10.24 μs
	512	40.96 μs
25 MHz	8	0.32 μs
	32	1.28 μs
	128	5.12 μs
	512	20.48 μs

Note Precision of time = __RIUSR_trcSW_base_time

2) __RIUSR_trcSW_init_tmr (Initialization function)

Description	This function initializes the hardware timer so that specification of __RIUSR_trcSW_read_cnt (Function to get time-stamp) may be realized. In the sample, this function initializes the CMT so that interruption may be generated, when the timer clock is counted 65536 times. This function is called-back from vsta_knl service call.
Parameter	None
Registers which do not need to guarantee	R1, R2, R3, R4, R5, R6, R7, R14, R15
PSW when started (Do not change)	PM = 0 (Supervisor mode) I = 0 (Disable all interrupts) U = 0 (System stack)
Available stack size	Up to 8 bytes

3) __RIUSR_trcSW_read_cnt (Function to get time-stamp)

Description	This function returns the elapsed time from the time of __RIUSR_trcSW_init_tmr (Initialization function) was called. The value returned must be in the range of from 0 and 0x7FFFFFFF, in units of the __RIUSR_trcSW_base_time (Time precision). In the sample, the lower 16 bits of the return value is CMT counter register, and the upper 16 bits is the number of the timer interruption. The return value must not be less than the previous return value.
Parameter	R5 (Out) : Elapsed time
Registers which do not need to guarantee	R3, R4
PSW when started (Do not change)	PM = 0 (Supervisor mode) I = 0 (Disable all interrupts) U = 0 (System stack)
Available stack size	Up to 8 bytes

4) Interrupt handler (Arbitrary function name)

<table>
<tr><td>Description</td><td>In the sample, this interrupt occurs when the timer clock is counted 65536 times.
This handler must not call service calls.
This handler exits by RTE instruction.
And this interrupt handler should be defined as follows in the system configuration file. Here, an example in case the vector number is 29 and function name is __RIUSR_trcSW_interrupt (assembly language level) is shown.
<pre>interrupt_vector[29] {
    entry_address = _RIUSR_trcSW_interrupt();
    os_int = NO;
};</pre></td></tr>
<tr><td>Parameter</td><td>None</td></tr>
<tr><td>Registers which do not need to guarantee</td><td>None</td></tr>
<tr><td>PSW when started (Do not change)</td><td>PM = 0 (Supervisor mode)
I = 0 (Disable all interrupts)
U = 0 (System stack)</td></tr>
<tr><td>Available stack size</td><td>Please take into consideration in “D.4 System Stack Size Estimation”.</td></tr>
</table>

15.3.2 Taking in long-statistics by software trace mode

In this mode, the RI600V4 gets time-stamp from user-own coding module. Usually, the hardware timer is used in order to generate time-stamp. The bit width of the counter of the hardware timer has necessity of 16 bits or more, and must be able to generate an interrupt when the timer clock is counted 65536 times. Note, CMT (Compare Match Timer) standardly built in RX family MCU is satisfying this requirement.
This section describes the specification of function and variables to be implemented as user-own coding module. Since each function does not follow ABI (Application Binary Interface) of the RX family C/C++ compiler, it needs to be implemented by using assembly language. In this section, function and variable name are described in assembly language level.

Note The sample file provided by the RI600V4 is “trcLONG_cmt.src”. This file uses CMT channel-1.

1) __RIUSR_trcLONG_base_time (TIme precision)

Define the unit of the time returned by __RIUSR_trcLONG_read_cnt (Function to get time-stamp) as a constant for the 32 bit- unsigned integer.
Usually, please set up the time of 1 count of hardware timer counter.
A typical setup in the case of using CMT is shown below.

PCLK	Dividing rate	Time precision of interrupt handler execution time (see Note 1)	Measurable maximum time of interrupt handler execution time (see Note 2)	Time precision of task execution time (see Note 3)	Measurable maximum time of task execution time (see Note 4)
12.5 MHz	8	0.64 μs	About 41 ms	5.12 μs	About 6 hr. 6 min.
	32	2.56 μs	About 167 ms	20.48 μs	About 24 hr. 26 min.
	128	10.24 μs	About 671 ms	81.92 μs	About 97 hr. 44 min.
	256	40.96 μs	About 2684 ms	327.68 μs	About 390 hr. 56 min.
25 MHz	8	0.32 μs	About 20 ms	2.56 μs	About 3 hr. 3 min.
	32	1.28 μs	About 83 ms	10.24 μs	About 12 hr. 13 min.
	128	5.12 μs	About 335 ms	40.96 μs	About 48 hr. 52 min.
	256	20.48 μs	About 1342 ms	163.84 μs	About 195 hr. 28 min.

Note 1 Time precision of interrupt handler execution time = __RIUSR_trcLONG_base_time

Note 2 Measurable maximum time of interrupt handler execution time = __RIUSR_trcLONG_base_time * 65536

Note 3 Time precision of task execution time = __RIUSR_trcLONG_base_time * 8

Note 4 Measurable maximum time of task execution time = __RIUSR_trcLONG_base_time * 8 * 0xFFFFFFFF

2) __RIUSR_trcLONG_timer_lvl (Interrupt priority level)

Define the interrupt priority level of the using hardware timer as a constant for the 8 bit- unsigned integer.
The execution time of interrupt handlers with interrupt priority level more than or equal to this interrupt priority level are not measured. The execution time of that interrupt handlers are appropriated for the execution time of the processing program (tasks, another interrupt handlers, or kernel idling) which was executing when that interrupt occurred.
The interrupt priority level of this timer recommends using the highest.

3) __RIUSR_trcLONG_init_tmr (Initialization function)

Description	This function initializes the hardware timer so that interruption which interrupt priority level is __RIUSR_trcLONG_timer_lvl (Interrupt priority level) may be generated, when the timer clock is counted 65536 times. This function is called-back from vsta_knl service call.
Parameter	None
Registers which do not need to guarantee	R1, R2, R3, R4, R5, R6, R7, R14, R15
PSW when started (Do not change)	PM = 0 (Supervisor mode) I = 0 (Disable all interrupts) U = 0 (System stack)
Available stack size	Up to 8 bytes

4) __RIUSR_trcLONG_read_cnt (Function to get time-stamp)

Description	This function returns the elapsed time from the previous interruption. The value returned must be in the range of from 0 and 65535 in units of the __RIUSR_trcLONG_base_time (TIme precision). In the sample, this function returns the value of the CMT counter register.
Parameter	R1 (Out) : Elapsed time
Registers which do not need to guarantee	R4
PSW when started (Do not change)	PM = 0 (Supervisor mode) I = 0 (Disable all interrupts) U = 0 (System stack)
Available stack size	Up to 8 bytes

5) Interrupt handler (Arbitrary function name)

Description	This handler should call RI600V4's __RI_trcLONG_update_time function.
	This handler must not call service calls.
	This handler exits by RTE instruction.
	And this interrupt handler should be defined as follows in the system configuration file. Here, an example in case the vector number is 29 and function name is __RIUSR_trcLONG_interrupt (assembly language level) is shown.
	`interrupt_vector[29] {`
	`    entry_address = _RIUSR_trcLONG_interrupt();`
	`    os_int = NO;`
	`};`

Parameter	None
Registers which do not need to guarantee	None
PSW when started (Do not change)	PM = 0 (Supervisor mode) I = 0 (Disable all interrupts) U = 0 (System stack)
Available stack size	Please take into consideration in "D.4 System Stack Size Estimation".

6) RI600V4's __RI_trcLONG_update_time function

This function is not user-own coding module, is implemented in the RI600V4. The following is a specification of this function.

Description	This function updates the current time information managed by RI600V4. This function should be called from the above interrupt handler.
Parameter	None
Registers which are not guaranteed	R1, R2
PSW when calling	PM = 0 (Supervisor mode) I = 0 (Disable all interrupts) U = 0 (System stack)
Stack size	0 bytes (It does not include 4 bytes which is used by BSR instruction for calling this function.)

15.4 Trace Buffer Size (Taking in Trace Chart by Software Trace Mode)

Table 15-2 shows the timing by which the trace buffer is consumed.

Table 15-2 Timing by which the trace buffer is consumed

Timing	Size to consume
Immediately after service call	12 bytes
Just before returning to application from RI600V4	8 bytes
When a task dispatches	8 bytes
When the RI600V4 enters Idling	8 bytes
When an interrupt handler starts	8 bytes
When an interrupt handler ends	8 bytes
When a cyclic handler starts	8 bytes
When a cyclic handler ends	8 bytes
When an alarm handler starts	8 bytes
When an alarm handler ends	8 bytes
When a task status changes	8 bytes

Table 15-3 shows the standard of measurable time.

Table 15-3 The standard of time after used up the buffers

Event generating frequency	Buffer Size			
	1 KB	4 KB	16 KB	64 KB
5 μs / Event	About 0.6 ms	About 2.4 ms	About 9.6 ms	About 38 ms
10 μs / Event	About 1.2 ms	About 4.8 ms	About 19 ms	About 77ms
50 μs / Event	About 6 ms	About 24 ms	About 96 ms	About 385ms
100 μs / Event	About 12 ms	About 48 ms	About 192 ms	About 771 ms
500 μs / Event	About 60 ms	About 240 ms	About 963 ms	About 3855 ms

15.5 Error of Total Execution Time

Total execution time of tasks or interrupt handlers is calculated by adding the execution time of each time. Therefore, the error of total execution time will also become large if the execution count increases.

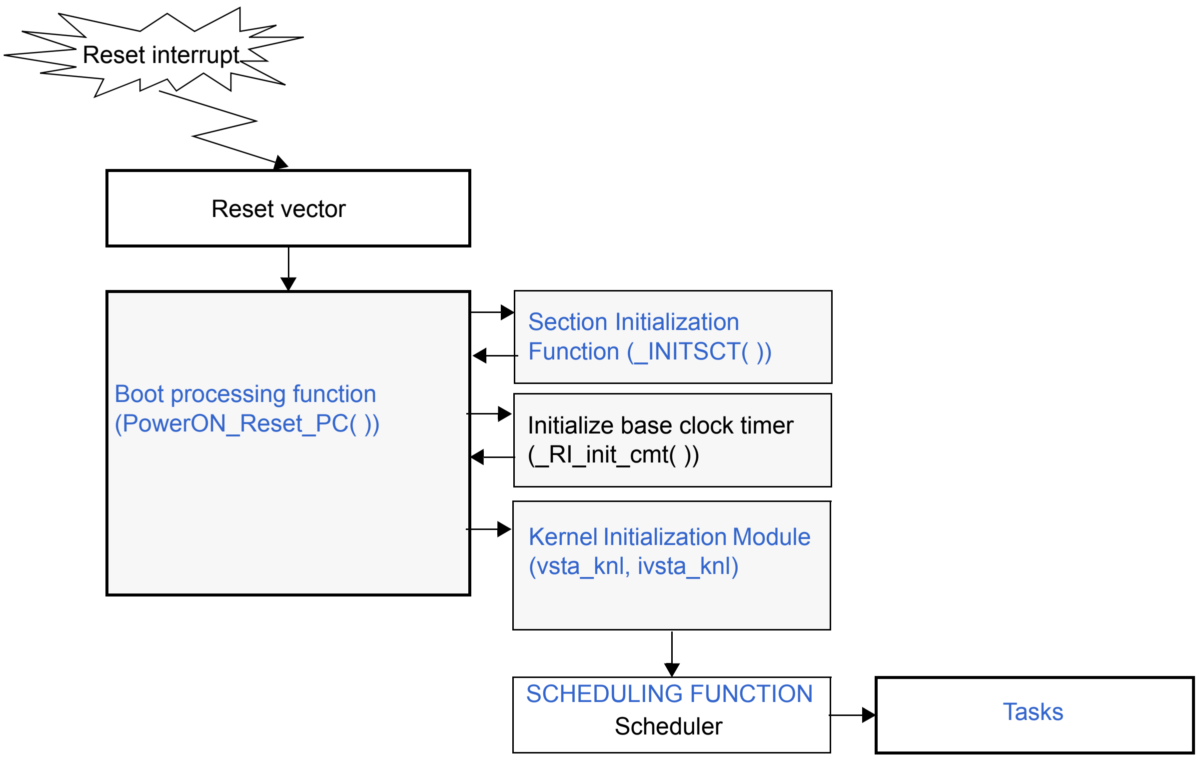
CHAPTER 16 SYSTEM INITIALIZATION

This chapter describes the system initialization routine performed by the RI600V4.

16.1 Outline

The following shows a processing flow from when a reset interrupt occurs until the control is passed to the task.

Figure 16-1 Processing Flow (System Initialization)



16.2 Boot Processing File (User-Own Coding Module)

The following should be described in the boot processing file.

1) Boot processing function (PowerON_Reset_PC( ))

2) System down routine (_RI_sys_dwn__)
For details, refer to "13.2.1 System down routine (_RI_sys_dwn__)".

3) Include kernel_ram.h and kernel_rom.h

Note The boot processing file which is provided by the RI600V4 as a sample file is "resetprg.c". This file includes System down routine (_RI_sys_dwn__).

16.2.1 Boot processing function (PowerON_Reset_PC( ))

The boot processing function is the program registered in the reset vector, and is executed in supervisor mode. Generally, following processing are required in the boot processing function.

- Initialize the processor and hardwares
  If using Fast Interrupt of the RX-MCU, initialize the FINTV register to the start address of the fast interrupt handler.
- Initialize C/C++ runtime environment (Initialize sections, etc.)
- Initialize base clock timer
  Call "void _RI_init_cmt(void)" which is defined in the "ri_cmt.h" generated by the cfg600.
  Refer to "8.9 Initialize Base Clock Timer".
- Start the RI600V4 (call vsta_knl or ivsta_knl)

- Basic form of boot processing function
  The boot processing function should be implemented as "void PowerON_Reset_PC(void)". When the name of the boot processing function is other, it is necessary to define the function name to "interrupt_fvector[31]" in the system configuration file.

Note For the details of the details of the static API "interrupt_fvector[]", refer to "19.19 Fixed Vector/Exception Vector Information (interrupt_fvector[])".

- The points of concern about the boot processing function
  - Stack
    Describe #pragma entry directive to be shown below. Thereby, the object code which sets the stack pointer (ISP) as the system stack at the head of the boot processing function is generated.

```
#pragma entry PowerON_Reset_PC
```

  - PSW register
    Keep the status that all interrupts are prohibited and in the supervisor mode until calling the Kernel Initialization Module (vsta_knl, ivsta_knl). This status is satisfied just behind CPU reset (PSW.I=0, PSW.PM=0). Generally, the boot processing function should not change the PSW.
  - EXTB register (RXv2 architecture)
    Initialize EXTB register to the start address of FIX_INTERRUPT_VECTOR section if needed. Please refer to "FIX_INTERRUPT_VECTOR section" in section 2.6.4.
  - Service call
    Since the boot processing function is executed before executing of Kernel Initialization Module (vsta_knl, ivsta_knl), service calls except vsta_knl and ivsta_knl must not be called from the boot processing function.

16.2.2 Include kernel_ram.h and kernel_rom.h

The boot processing file must include “kernel_ram.h” and “kernel_rom.h”, which are generated by the cfg600, in this order.

16.2.3 Compiler option for boot processing file

The following compiler options are required for the boot processing file.

- “-lang=c” or “-lang=c99”
- “-nostuff”
- Suitable “-isa” or “-cpu”

Note Compiler option “-isa” is supported by the compiler CC-RX V2.01 or later.

16.2.4 Example of the boot processing file

```
#include    <machine.h>
#include    <_h_c_lib.h>
//#include  <stddef.h>                    // Remove the comment when you use errno
//#include  <stdlib.h>                    // Remove the comment when you use rand()
#include    "typedefine.h"                // Define Types
#include    "kernel.h"                    // Provided by RI600V4
#include    "kernel_id.h"                 // Generated by cfg600

#if (((_RI_CLOCK_TIMER) >=0) && ((_RI_CLOCK_TIMER) <= 3))
#include    "ri_cmt.h"    // Generated by cfg600
                    // Do comment-out when clock.timer is either NOTIMER or OTHER.
#endif

#ifdef __cplusplus
extern "C" {
#endif
void PowerON_Reset_PC(void);
void main(void);
#ifdef __cplusplus
}
#endif

//#ifdef __cplusplus                // Use SIM I/O
//extern "C" {
//#endif
//extern void _INIT_IOLIB(void);
//extern void _CLOSEALL(void);
//#ifdef __cplusplus
//}
//#endif

#define FPSW_init 0x00000000    // FPSW bit base pattern

//extern void srand(_UINT);     // Remove the comment when you use rand()
//extern _SBYTE *_s1ptr;        // Remove the comment when you use strtok()

//#ifdef __cplusplus            // Use Hardware Setup
//extern "C" {
//#endif
//extern void HardwareSetup(void);
```

```
//#ifdef __cplusplus
//}
//#endif

//#ifdef __cplusplus   // Remove the comment when you use global class object
//extern "C" {         // Sections C$INIT and C$END will be generated
//#endif
//extern void _CALL_INIT(void);
//extern void _CALL_END(void);
//#ifdef __cplusplus
//}
//#endif

#pragma section ResetPRG        // output PowerON_Reset to PResetPRG section

////////////////////////////////////////////////////////////////////////////
// Boot processing
////////////////////////////////////////////////////////////////////////////
#pragma entry PowerON_Reset_PC

void PowerON_Reset_PC(void)
{

#ifdef __ROZ                    // Initialize FPSW
#define _ROUND 0x00000001           // Let FPSW RMbits=01 (round to zero)
#else
#define _ROUND 0x00000000           // Let FPSW RMbits=00 (round to nearest)
#endif
#ifdef __DOFF
#define _DENOM 0x00000100           // Let FPSW DNbit=1 (denormal as zero)
#else
#define _DENOM 0x00000000           // Let FPSW DNbit=0 (denormal as is)
#endif

//    set_extb(__sectop("FIX_INTERRUPT_VECTOR"));// Initialize EXTB register
                                                 //   (only for RXv2 arch.)
    set_fpsw(FPSW_init | _ROUND | _DENOM);

    _INITSCT();

//  _INIT_IOLIB();                  // Use SIM I/O

//  errno=0;                        // Remove the comment when you use errno
//  srand((_UINT)1);                // Remove the comment when you use rand()
//  _s1ptr=NULL;                    // Remove the comment when you use strtok()

//  HardwareSetup();                // Use Hardware Setup
    nop();

//  set_fintv(<handler address>);   // Initialize FINTV register

#if (((_RI_CLOCK_TIMER) >=0) && ((_RI_CLOCK_TIMER) <= 3))
    _RI_init_cmt();  // Initialize CMT for RI600V4
            // Do comment-out when clock.timer is either NOTIMER or OTHER.
#endif

//  _CALL_INIT();           // Remove the comment when you use global class object

    vsta_knl();                     // Start RI600V4
                                    // Never return from vsta_knl
```

```
//   _CLOSEALL();                        // Use SIM I/O

//   _CALL_END();            // Remove the comment when you use global class object

    brk();
}

//////////////////////////////////////////////////////////////////////////////
// System down routine for RI600V4
//////////////////////////////////////////////////////////////////////////////
#pragma section P PRI_KERNEL
#pragma section B BRI_RAM
struct SYSDWN_INF{
    W   type;
    VW  inf1;
    VW  inf2;
    VW  inf3;
};

volatile struct SYSDWN_INF _RI_sysdwn_inf;

void _RI_sys_dwn__( W type, VW inf1, VW inf2, VW inf3 )
{
   // Now PSW.I=0 (all interrupts are masked.)
    _RI_sysdwn_inf.type = type;
    _RI_sysdwn_inf.inf1 = inf1;
    _RI_sysdwn_inf.inf2 = inf2;
    _RI_sysdwn_inf.inf3 = inf3;

   while(1)
   ;
}

//////////////////////////////////////////////////////////////////////////////
// RI600V4 system data
//////////////////////////////////////////////////////////////////////////////
#include "kernel_ram.h"     // generated by cfg600
#include "kernel_rom.h"     // generated by cfg600
```

16.3 Kernel Initialization Module (vsta_knl, ivsta_knl)

The kernel initialization module is executed by calling vsta_knl, ivsta_knl. Generally, vsta_knl, ivsta_knl is called from the Boot processing function (PowerON_Reset_PC( )).
The following processing is executed in the kernel initialization module.

1) Initialize ISP register to the end address of SI section + 1

2) Initialize INTB register to the start address of the relocatable vector table (INTERRUPT_VECTOR section). The relocatable vector table is generated by the cfg600.

3) Initialize the system time to 0.

4) Create various object which are defined in the system configuration file.

5) Pass control to scheduler

16.4 Section Initialization Function (_INITSCT( ))

The section initialization function "_INITSCT()" called from Boot processing function (PowerON_Reset_PC( )) is provided by the compiler. The _INITSCT() clears the uninitialized data section to 0 and initializes the initialized data section in order to the tables described in the Section information file (User-Own Coding Module).
The user needs to write the sections to be initialized to the tables for section initialization (DTBL and BTBL) in the section information file. The section address operator is used to set the start and end addresses of the sections used by the _INITSCT(). Section names in the section initialization tables are declared, using C$BSEC for uninitialized data areas, and C$DSEC for initialized data areas.
Initialized sections written in DTBL must be mapped from ROM to RAM by using "-rom" linker option. For details, refer to "2.6.5 Initialized data section".

Note See "CubeSuite+ Integrated Development Environment User's Manual: RX Coding" for details of the _INITSCT().

16.4.1 Section information file (User-Own Coding Module)

The section information file should be implemented as user-own coding module.
The example of the section information file is shown below.

Note The section information file which is provided by the RI600V4 as a sample file is "dbsct.c".

```
#include "typedefine.h"

#pragma unpack

#pragma section C C$DSEC
extern const struct {
    _UBYTE *rom_s;      /* Start address of the initialized data section in ROM */
    _UBYTE *rom_e;      /* End address of the initialized data section in ROM   */
    _UBYTE *ram_s;      /* Start address of the initialized data section in RAM */
}   _DTBL[] = {
    { __sectop("D"), __secend("D"), __sectop("R") },
    { __sectop("D_2"), __secend("D_2"), __sectop("R_2") },
    { __sectop("D_1"), __secend("D_1"), __sectop("R_1") },
    /* RI600V4 section */
    { __sectop("DRI_ROM"), __secend("DRI_ROM"), __sectop("RRI_RAM") }
};
#pragma section C C$BSEC
extern const struct {
    _UBYTE *b_s;        /* Start address of non-initialized data section */
    _UBYTE *b_e;        /* End address of non-initialized data section */
}   _BTBL[] = {
    { __sectop("B"), __secend("B") },
    { __sectop("B_2"), __secend("B_2") },
    { __sectop("B_1"), __secend("B_1") }
};

#pragma section

/*
** CTBL prevents excessive output of L1100 messages when linking.
** Even if CTBL is deleted, the operation of the program does not change.
*/
_UBYTE * const _CTBL[] = {
    __sectop("C_1"), __sectop("C_2"), __sectop("C"),
    __sectop("W_1"), __sectop("W_2"), __sectop("W")
};

#pragma packoption
```

16.5 Registers in Fixed Vector Table/Exception Vector table

For some MCUs, the endian select register, ID code protection on connection of the on-chip debugger, etc. are assigned in the address from 0xFFFFFF80 to 0xFFFFFFBF in fixed vector table (RXv1 architecture) / exception vector table (RXv2 architecture). To set up such registers, describe “interrupt_fvector[]” in the system configuration file. For details, refer to “19.19 Fixed Vector/Exception Vector Information (interrupt_fvector[])”.

CHAPTER 17 DATA TYPES AND MACROS

This chapter describes the data types and macros, which are used when issuing service calls provided by the RI600V4.

Note <ri_root> indicates the installation folder of RI600V4.
The default folder is “C:\Program Files\Renesas Electronics\CubeSuite+\RI600V4”.

17.1 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 <ri_root>\in600\kernel.h, or <ri_root>\inc600\itron.h that is included by kernel.h.

Table 17-1 Data Types

Macro	Data Type	Description
B	signed char	Signed 8-bit integer
H	signed short	Signed 16-bit integer
W	signed long	Signed 32-bit integer
D	signed long long	Signed 64-bit integer
UB	unsigned char	Unsigned 8-bit integer
UH	unsigned short	Unsigned 16-bit integer
UW	unsigned long	Unsigned 32-bit integer
UD	unsigned long long	Unsigned 64-bit integer
VB	signed char	8-bit value with unknown data type
VH	signed short	16-bit value with unknown data type
VW	signed long	32-bit value with unknown data type
VD	signed long long	64-bit value with unknown data type
VP	void *	Pointer to unknown data type
FP	void (*)	Processing unit start address (pointer to a function)
INT	signed long	Signed 32-bit integer
UINT	unsigned long	Unsigned 32-bit integer
BOOL	signed long	Boolean value (TRUE or FALSE)
ER	signed long	Error code
ID	signed short	Object ID
ATR	unsigned short	Object attribute
STAT	unsigned short	Object state
MODE	unsigned short	Service call operational mode
PRI	signed short	Priority for tasks or messages
SIZE	unsigned long	Memory area size (in bytes)
TMO	signed long	Time-out (in millisecond)
RELTIM	unsigned long	Relative time (in millisecond)

Macro	Data Type	Description
VP_INT	signed long	Pointer to unknown data type, or signed 32-bit integer
ER_UINT	signed long	Error code, or signed 32-bit integer
FLGPTN	unsigned long	Bit pattern of eventflag
IMASK	unsigned short	Interrupt mask level

17.2 Macros

This section explains the macros (for current state, processing program attributes, or the like) used when issuing a service call provided by the RI600V4.

17.2.1 Constant macros

The following lists the constant macros.
The constant macros are defined by either of following header files.

- <ri_root>\inc600\kernel.h
- <ri_root>\inc600\itron.h, which s included by kernel.h
- System information header file kernel_id.h, which is generated by the cfg600.
  The contents of this file is changed according to the system configuration file.

Table 17-2 Constant Macros

Classification	Macro	Definition	Where	Description
General	NULL	0	itron.h	Null pointer
	TRUE	1	itron.h	True
	FALSE	0	itron.h	False
	E_OK	0	itron.h	Normal completion
Attribute	TA_NULL	0	itron.h	Object attribute unspecified
	TA_TFIFO	0x0000	kernel.h	Task wait queue in FIFO order
	TA_TPRI	0x0001	kernel.h	Task wait queue is managed in task current priority order. Among tasks with the same priority, they are queued in FIFO order.
	TA_MFIFO	0x0000	kernel.h	Message queue in FIFO order
	TA_MPRI	0x0002	kernel.h	Message queue is managed in message priority order. Among messages with the same priority, they are queued in FIFO order.
	TA_ACT	0x0002	kernel.h	Task is activated after creation
	TA_WSGL	0x0000	kernel.h	Do not allow multiple tasks to wait for eventflag
	TA_WMUL	0x0002	kernel.h	Allow multiple tasks to wait for eventflag
	TA_CLR	0x0004	kernel.h	Clear eventflag when freed from WAITING state
	TA_CEILING	0x0003	kernel.h	Priority ceiling protocol
	TA_STA	0x0002	kernel.h	Create cyclic hander in operational state
	TA_PHS	0x0004	kernel.h	Save cyclic hander phase
Time-out	TMO_POL	0	itron.h	Polling
	TMO_FEVR	-1	itron.h	Waiting forever

Classification	Macro	Definition	Where	Description
Operation mode	TWF_ANDW	0x0000	kernel.h	Eventflag AND wait
	TWF_ORW	0x0001	kernel.h	Eventflag OR wait
Object state	TTS_RUN	0x0001	kernel.h	RUNNING state
	TTS_RDY	0x0002	kernel.h	READY state
	TTS_WAI	0x0004	kernel.h	WAITING state
	TTS_SUS	0x0008	kernel.h	SUSPENDED state
	TTS_WAS	0x000C	kernel.h	WAITING-SUSPENDED state
	TTS_DMT	0x0010	kernel.h	DORMANT state
	TTW_SLP	0x0001	kernel.h	Sleeping state
	TTW_DLY	0x0002	kernel.h	Delayed state
	TTW_SEM	0x0004	kernel.h	Waiting state for a semaphore resource
	TTW_FLG	0x0008	kernel.h	Waiting state for an eventflag
	TTW_SDTQ	0x0010	kernel.h	Sending waiting state for a data queue
	TTW_RDTQ	0x0020	kernel.h	Receiving waiting state for a data queue
	TTW_MBX	0x0040	kernel.h	Receiving waiting state for a mailbox
	TTW_MTX	0x0080	kernel.h	Waiting state for a mutex
	TTW_SMBF	0x0100	kernel.h	Sending waiting state for a message buffer
	TTW_RMBF	0x0200	kernel.h	Receiving waiting state for a message buffer
	TTW_MPF	0x2000	kernel.h	Waiting state for a fixed-sized memory block
	TTW_MPL	0x4000	kernel.h	Waiting state for a variable-sized memory block
	TCYC_STP	0x0000	kernel.h	Cyclic handler in non-operational state
	TCYC_STA	0x0001	kernel.h	Cyclic handler in operational state
	TALM_STP	0x0000	kernel.h	Alarm handler in non-operational state
	TALM_STA	0x0001	kernel.h	Alarm handler in operational state
Others	TSK_SELF	0	kernel.h	Specify invoking task
	TSK_NONE	0	kernel.h	No relevant task
	TPRI_SELF	0	kernel.h	Specify base priority of invoking task
	TPRI_INI	0	kernel.h	Specify initial priority

Classification	Macro	Definition	Where	Description
Kernel configuration	TMIN_TPRI	1	kernel.h	Minimum task priority
	TMAX_TPRI	system.priority	kernel_id.h	Maximum task priority
	TMIN_MPRI	1	kernel.h	Minimum message priority
	TMAX_MPRI	system.message_pri	kernel_id.h	Maximum message priority
	TKERNEL_MAKER	0x011B	kernel.h	Kernel maker code
	TKERNEL_PRID	0x0003	kernel.h	Identification number of the kernel
	TKERNEL_SPVER	0x5403	kernel.h	Version number of the ITRON specification
	TKERNEL_PRVER	0x0130	kernel.h	Version number of the kernel
	TMAX_ACTCNT	255	kernel.h	Maximum number of queued task activation requests
	TMAX_WUPCNT	255	kernel.h	Maximum number of queued task wake-up requests
	TMAX_SUSCNT	1	kernel.h	Maximum number of nested task suspension requests
	TBIT_FLGPTN	32	kernel.h	Number of bits in an eventflag
	TIC_NUME	system.tic_nume	kernel_id.h	Numerator of base clock interval
	TIC_DENO	system.tic_deno	kernel_id.h	Denominator of base clock interval
	TMAX_MAXSEM	65535	kernel.h	Maximum value of the maximum semaphore resource count
	VTMAX_TSK	Number of “task[]”s	kernel_id.h	Maximum task ID
	VTMAX_SEM	Number of “semaphore[]”s	kernel_id.h	Maximum semaphore ID
	VTMAX_FLG	Number of “flag[]”s	kernel_id.h	Maximum eventflag ID
	VTMAX_DTQ	Number of “dataqueue[]”s	kernel_id.h	Maximum data queue ID
	VTMAX_MBX	Number of “mailbox[]”s	kernel_id.h	Maximum mailbox ID
	VTMAX_MTX	Number of “mutex[]”s	kernel_id.h	Maximum mutex ID
	VTMAX_MBF	Number of “message_buffer[]”s	kernel_id.h	Maximum message buffer ID
	VTMAX_MPF	Number of “memorypool[]”s	kernel_id.h	Maximum fixed-sized memory pool ID
	VTMAX_MPL	Number of “variable_memorypool[]”s	kernel_id.h	Maximum variable-sized memory pool ID
	VTMAX_CYH	Number of “cyclic_hand[]”s	kernel_id.h	Maximum cyclic handler ID
	VTMAX_ALH	Number of “alarm_hand[]”s	kernel_id.h	Maximum alarm handler ID
	VTSZ_MBFTBL	4	kernel.h	Size of message buffer's message management table (in bytes)
	VTMAX_AREASIZE	0x10000000	kernel.h	Maximum size of various areas (in bytes)
	VTKNL_LVL	system.system_IPL	kernel_id.h	Kernel interrupt mask level
	VTIM_LVL	clock.IPL	kernel_id.h	Base clock interrupt level

Classification	Macro	Definition	Where	Description
Error code	E_NOSPT	-9	itron.h	Unsupported function
	E_PAR	-17	itron.h	Parameter error
	E_ID	-18	itron.h	Invalid ID number
	E_CTX	-25	itron.h	Context error
	E_ILUSE	-28	itron.h	Illegal use of service call
	E_OBJ	-41	itron.h	Object state error
	E_QOVR	-43	itron.h	Queuing overflow
	E_RLWAI	-49	itron.h	Forced release from WAITING state
	E_TMOUT	-50	itron.h	Polling failure of time-out
	EV_RST	-127	itron.h	Released from WAITING state by the object reset

17.2.2 Function Macros

The following lists the function macros.
The function macros are defined by <ri_root>\inc600\itron.h.

1) ER MERCD (ER *ercd*)
Return the main error code of *ercd*.

2) ER SERCD (ER *ercd*)
Return sub error code of *ercd*.

3) ER ERCD (ER *mercd*, ER *sercd*)
Return the error code from the main error code indicated by *mercd* and sub error code indicated by *sercd*.

Note In the error code returned from the RI600V4, all sub error code is -1, and all main error code is same as the value described in Table 17-2.

CHAPTER 18 SERVICE CALLS

This chapter describes the service calls supported by the RI600V4.

18.1 Outline

The service calls provided by the RI600V4 are service routines provided for indirectly manipulating the resources (tasks, semaphores, etc.) managed by the RI600V4 from a processing program.
The service calls provided by the RI600V4 are listed below by management module.

- Task management functions

act_tsk	iact_tsk	can_act	ican_act
sta_tsk	ista_tsk	ext_tsk	ter_tsk
chg_pri	ichg_pri	get_pri	iget_pri
ref_tsk	iref_tsk	ref_tst	iref_tst

- 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)

wai_sem	pol_sem	ipol_sem	twai_sem
sig_sem	isig_sem	ref_sem	iref_sem

- Synchronization and communication functions (eventflags)

set_flg	iset_flg	clr_flg	iclr_flg
wai_flg	pol_flg	ipol_flg	twai_flg
ref_flg	iref_flg		

- Synchronization and communication functions (data queues)

snd_dtq	psnd_dtq	ipsnd_dtq	tsnd_dtq
fsnd_dtq	ifsnd_dtq	rcv_dtq	prcv_dtq
iprcv_dtq	trcv_dtq	ref_dtq	iref_dtq

- Synchronization and communication functions (mailboxes)

snd_mbx	isnd_mbx	rcv_mbx	prcv_mbx
iprcv_mbx	trcv_mbx	ref_mbx	iref_mbx

- Extended synchronization and communication functions (mutexes)

loc_mtx	ploc_mtx	tloc_mtx	unl_mtx
ref_mtx			

- Extended synchronization and communication functions (message buffers)

snd_mbf	psnd_mbf	ipsnd_mbf	tsnd_mbf
rcv_mbf	prcv_mbf	trcv_mbf	ref_mbf
iref_mbf			

- Memory pool management functions (fixed-sized memory pools)

get_mpf	pget_mpf	ipget_mpf	tget_mpf
rel_mpf	irel_mpf	ref_mpf	iref_mpf

- Memory pool management functions (variable-sized memory pools)

get_mpl	pget_mpl	ipget_mpl	tget_mpl
rel_mpl	ref_mpl	iref_mpl	

- Time management functions

set_tim	iset_tim	get_tim	iget_tim
sta_cyc	ista_cyc	stp_cyc	istp_cyc
ref_cyc	iref_cyc	sta_alm	ista_alm
stp_alm	istp_alm	ref_alm	iref_alm

- System state management functions

rot_rdq	irot_rdq	get_tid	iget_tid
loc_cpu	iloc_cpu	unl_cpu	iunl_cpu
dis_dsp	ena_dsp	sns_ctx	sns_loc
sns_dsp	sns_dpn	vsys_dwn	ivsys_dwn
vsta_knl	ivsta_knl		

- Interrupt management functions

chg_ims	ichg_ims	get_ims	iget_ims

- System configuration management functions

ref_ver	iref_ver		

- Object reset functions

vrst_dtq	vrst_mbx	vrst_mbf	vrst_mpf
vrst_mpl			

18.1.1 Method for calling service calls

The service calls can be calls by the same way as normal C-language function.

Note To call the service calls provided by the RI600V4 from a processing program, the header files listed below must be coded (include processing).

kernel.h: Standard header file

kernel_id.h System information header file, which is generated by the cfg600

18.2 Explanation of Service Call

The following explains the service calls supported by the RI600V4, in the format shown below.

1)

2) → **Outline**

3) → **C format**

4) → **Parameter(s)**

I/O	Parameter	Description

5) → **Explanation**

6) → **Return value**

Macro	Value	Description

1) Name
Indicates the name of the service call.

2) Outline
Outlines the functions of the service call.

3) C format
Indicates the format to be used when describing a service call to be issued in C language.

4) Parameter(s)
Service call parameters are explained in the following format.

I/O	Parameter	Description
A	B	C

A) Parameter classification
I: Parameter input to RI600V4.
O: Parameter output from RI600V4.

B) Parameter data type

C) Description of parameter

5) Explanation
Explains the function of a service call.

6) Return value
Indicates a service call's return value using a macro and value.

Macro	Value	Description
A	B	C

A) Macro of return value

B) Value of return value

C) Description of return value

18.2.1 Task management functions

The following shows the service calls provided by the RI600V4 as the task management functions.

Table 18-1 Task Management Functions

Service Call	Function	Useful Range
act_tsk	Activate task (queues an activation request)	Task
iact_tsk	Activate task (queues an activation request)	Non-task
can_act	Cancel task activation requests	Task
ican_act	Cancel task activation requests	Non-task
sta_tsk	Activate task (does not queue an activation request)	Task
ista_tsk	Activate task (does not queue an activation request)	Non-task
ext_tsk	Terminate invoking task	Task
ter_tsk	Terminate task	Task
chg_pri	Change task priority	Task
ichg_pri	Change task priority	Non-task
get_pri	Reference task current priority	Task
iget_pri	Reference task current priority	Non-task
ref_tsk	Reference task state	Task
iref_tsk	Reference task state	Non-task
ref_tst	Reference task state (simplified version)	Task
iref_tst	Reference task state (simplified version)	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);
```

Parameter(s)

I/O	Parameter	Description
I	ID *tskid;*	ID number of the task. TSK_SELF: Invoking task. Value: ID number of the task.

Explanation

These service calls move the 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 RI600V4.
At this time, the following processing is done.

Table 18-2 Processing Performed at Task Activation

No.	Content of processing
1	Initializes the task's base priority and current priority.
2	Clears the number of queued walk-up requests.
3	Clears the number of nested suspension count

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 1 to the activation request counter).

Note 1 The activation request counter managed by the RI600V4 is configured in 8-bit widths. If the number of activation requests exceeds the maximum count value 255 as a result of issuing this service call, the counter manipulation processing is therefore not performed but “E_QOVR” is returned.

Note 2 Extended information specified in Task Information (task[]) is passed to the task activated by issuing these service calls.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *tskid* < 0 - *tskid* > VTMAX_TSK - When iact_tsk was issued from a non-task, TSK_SELF was specified for *tskid*.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - The iact_tsk was issued from task. - The act_tsk was issued from non-task. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_QOVR	-43	Queue overflow. - Activation request count exceeded 255.

can_act
ican_act

Outline

Cancel task activation requests.

C format

```
ER_UINT can_act (ID tskid);
ER_UINT ican_act (ID tskid);
```

Parameter(s)

I/O	Parameter	Description
I	ID *tskid*;	ID number of the task. TSK_SELF: Invoking task. Value: ID number of the task.

Explanation

This service call cancels all of the activation requests queued to the task specified by parameter *tskid* (sets the activation request counter to 0).
When this service call is terminated normally, the number of cancelled activation requests is returned.

Note This service call does not perform status manipulation processing but performs the setting of activation request counter. Therefore, the task does not move from a state such as the READY state to the DORMANT state.

Return value

Macro	Value	Description
E_ID	-18	Invalid ID number. - *tskid* < 0 - *tskid* > VTMAX_TSK - When the iact_tsk was issued from a non-task, TSK_SELF was specified for *tskid*.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”. Note When the ican_act is issued from task or the can_act is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.

Macro	Value	Description
-	0	Normal completion. - Activation request count is 0. - Specified task is in the DORMANT state.
-	Positive value	Normal completion (activation request count).

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);
```

Parameter(s)

I/O	Parameter	Description
I	`ID tskid;`	ID number of the task.
I	`VP_INT stacd;`	Start code of the task.

Explanation

These service calls move the 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 RI600V4.
At this time, processing described in Table 18-2 is done.
These service calls do not perform queuing of activation requests. If the target task is in a state other than the DORMANT state, the status manipulation processing for the target task is therefore not performed but “E_OBJ” is returned.
The *stacd* is passed to the target task.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *tskid* ≤ 0 - *tskid* > VTMAX_TSK
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - The ista_tsk was issued from task. - The sta_tsk was issued from non-task. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.

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);
```

Parameter(s)

None.

Explanation

This service call moves the 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 RI600V4 scheduling subject.
At this time, the following processing is done.

Table 18-3 Processing Performed at Task Termination

No.	Content of processing
1	Unlocks the mutexes which are locked by the terminated task. (processing equivalent to unl_mtx will be executed)

The CPU locked state and dispatching disabled state is cancelled.
If an activation request has been queued to the invoking task (the activation request counter > 0) when this service call is issued, this service call moves the task from the RUNNING state to the DORMANT state, decrements the activation request counter (by subtracting 1 from the activation request counter), and then moves the task from the DORMANT state to the READY state. At this time, processing described in Table 18-2 is done.
This service call does not return. In the following cases, this service call causes SYSTEM DOWN.

- This service call was issued from non-task.
- This service call was issued in the status “PSW.IPL > kernel interrupt mask level”

Note 1 When the return instruction is issued in the task entry function, the same processing as ext_tsk is performed.

Note 2 This service call does not have the function to automatically free the resources except the mutex hitherto occupied by the task (e.g., semaphores and memory blocks). Make sure the task frees these resources before it terminates

Return value

None.

ter_tsk

Outline

Terminate task.

C format

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

Parameter(s)

I/O	Parameter	Description
I	`ID tskid;`	ID number of the task.

Explanation

This service call forcibly moves the task specified by parameter *tskid* to the DORMANT state.
As a result, the target task is excluded from the RI600V4 scheduling subject.
At this time, processing described in Table 18-3 is done.
If an activation request has been queued to the target task (the activation request counter > 0) when this service call is issued, this service call moves the task to the DORMANT state, decrements the activation request counter (by subtracting 1 from the activation request counter), and then moves the task from the DORMANT state to the READY state. At this time, processing described in Table 18-2 is done.

Note This service call does not have the function to automatically free the resources except the mutex hitherto occupied by the task (e.g., semaphores and memory blocks). Make sure the task frees these resources before it terminates

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *tskid* $\leq$ 0 - *tskid* > VTMAX_TSK
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued from non-task. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_ILUSE	-28	Illegal service call use. - Specified task is the invoking task.

Macro	Value	Description
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);
```

Parameter(s)

I/O	Parameter	Description
I	`ID tskid;`	ID number of the task. TSK_SELF: Invoking task. Value: ID number of the task.
I	`PRI tskpri;`	New base priority of the task. TPRI_INI: Initial priority. Value: New base priority of the task.

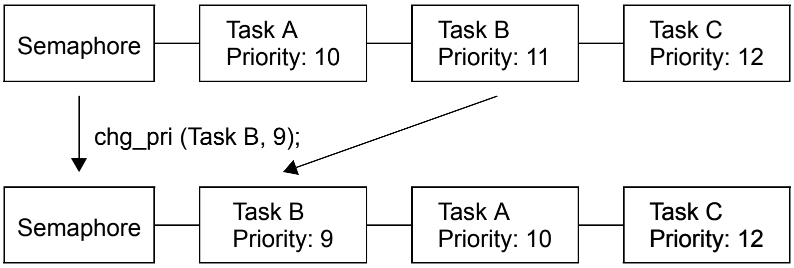
Explanation

This service call changes the base priority of the task specified by parameter *tskid* to a value specified by parameter *tskpri*.
The changed base priority is effective until the task terminates or this service call is issued. When next the task is activated, the base priority is the initial priority which is specified at the task creation.
This service call also changes the current priority of the target task to a value specified by parameter *tskpri*. However, the current priority is not changed when the target task has locked mutexes.
If the target task has locked mutexes or is waiting for mutex to be locked and if *tskpri* is higher than the ceiling priority of either of the mutexes, this service call returns “E_ILUSE”.
When the current priority is changed, the following state variations are generated.

1) When the target task is in the RUNNING or READY state.
This service call re-queues the task at the end of the ready queue corresponding to the priority specified by parameter *tskpri*.

2) When the target task is queued to a wait queue of the object with TA_TPRI or TA_CEILING attribute.
This service call re-queues the task to the wait queue corresponding to the priority specified by parameter *tskpri*. When two or more tasks of same current priority as *tskpri*, this service call re-queues the target task at the end among their tasks.

Example When three tasks (task A: priority level 10, task B: priority level 11, task C: priority level 12) are queued to the semaphore wait queue in the order of priority, and the priority level of task B is changed from 11 to 9, the wait order will be changed as follows.



Note For current priority and base priority, refer to "6.2.2 Current priority and base priority".

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *tskpri* < 0 - *tskpri* > TMAX_TPRI
E_ID	-18	Invalid ID number. - *tskid* < 0 - *tskid* > VTMAX_TSK - When ichg_pri was issued from a non-task, TSK_SELF was specified for *tskid*.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - The ichg_pri was issued from task. - The chg_pri was issued from non-task. - This service call was issued in the status "PSW.IPL > kernel interrupt mask level".
E_ILUSE	-28	Illegal use of service call. - *tskpri* < The ceiling priority of the mutex locked by the target task. - *tskpri* < The ceiling priority of the mutex by which the target task waits for lock.
E_OBJ	-41	Object state error. - Specified task is in the DORMANT state.

get_pri
iget_pri

Outline

Reference task current priority.

C format

```
ER      get_pri (ID tskid, PRI *p_tskpri);
ER      iget_pri (ID tskid, PRI *p_tskpri);
```

Parameter(s)

I/O	Parameter	Description
I	`ID tskid;`	ID number of the task. TSK_SELF: Invoking task. Value: ID number of the task.
O	`PRI *p_tskpri;`	Pointer to the area returning the current priority of the task.

Explanation

This service call stores the current priority of the task specified by parameter *tskid* in the area specified by parameter *p_tskpri*.

Note For current priority and base priority, refer to "6.2.2 Current priority and base priority".

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *tskid* < 0 - *tskid* > VTMAX_TSK - When this service call was issued from a non-task, TSK_SELF was specified for *tskid*.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status "PSW.IPL > kernel interrupt mask level". Note When the iget_pri is issued from task or the get_pri is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.

Macro	Value	Description
E_OBJ	-41	Object state error. - Specified task is in the DORMANT state.

ref_tsk
iref_tsk

Outline

Reference task state.

C format

```
ER      ref_tsk (ID tskid, T_RTSK *pk_rtsk);
ER      iref_tsk (ID tskid, T_RTSK *pk_rtsk);
```

Parameter(s)

I/O	Parameter	Description
I	ID tskid;	ID number of the task. TSK_SELF: Invoking task. Value: ID number of the task.
O	T_RTSK *pk_rtsk;	Pointer to the packet returning the task state.

[Task state packet: T_RTSK]

```
typedef struct  t_rtsk {
    STAT    tskstat;        /*Current state*/
    PRI     tskpri;         /*Current priority*/
    PRI     tskbpri;        /*Base priority*/
    STAT    tskwait;        /*Reason for waiting*/
    ID      wobjid;         /*Object ID number for which the task is waiting*/
    TMO     lefttmo;        /*Remaining time until time-out*/
    UINT    actcnt;         /*Activation request count*/
    UINT    wupcnt;         /*Wake-up request count*/
    UINT    suscnt;         /*Suspension count*/
} T_RTSK;
```

Explanation

Stores task state packet (current state, current priority, etc.) of the task specified by parameter *tskid* in the area specified by parameter *pk_rtsk*.

- *tskstat*
  Stores the current state.

  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*
  Stores the current priority.
  The *tskpri* is effective only when the *tskstat* is other than TTS_DMT.

- *tskbpri*
  Stores the base priority.
  The *tskbpri* is effective only when the *tskstat* is other than TTS_DMT.

- *tskwait*
  Stores the reason for waiting.
  The *tskwait* is effective only when the *tskstat* is TTS_WAI or TTS_WAS.

TTW_SLP:	Sleeping state caused by slp_tsk or tslp_tsk
TTW_DLY:	Delayed state caused by dly_tsk
TTW_SEM:	WAITING state for a semaphore resource caused by wai_sem or twai_sem
TTW_FLG:	WAITING state for an eventflag caused by wai_flg or twai_flg
TTW_SDTQ:	Sending WAITING state for a data queue caused by snd_dtq or tsnd_dtq
TTW_RDTQ:	Receiving WAITING state for a data queue caused by rcv_dtq or trcv_dtq
TTW_MBX:	Receiving WAITING state for a mailbox caused by rcv_mbx or trcv_mbx
TTW_MTX:	WAITING state for a mutex caused by loc_mtx or tloc_mtx
TTW_SMBF:	Sending WAITING state for a message buffer caused by snd_mbf or tsnd_mbf
TTW_RMBF:	Receiving WAITING state for a message buffer caused by rcv_mbf or trcv_mbf
TTW_MPF:	WAITING state for a fixed-sized memory block caused by get_mpf or tget_mpf
TTW_MPL:	WAITING state for a variable-sized memory block caused by get_mpl or tget_mpl

- *wobjid*
  Stores the object (such as semaphore, eventflag, etc.) ID number for which the task waiting.
  The *wobjid* is effective only when the *tskwait* is TTW_SEM or TTW_FLG or TTW_SDTQ or TTW_RDTQ or TTW_MBX or TTW_MTX or TTW_SMBF or TTW_RMBF or TTW_MPF or TTW_MPL.

- *lefttmo*
  Stores the remaining time until time-out (in millisecond).
  The TMO_FEVR is stored for waiting forever.
  The *lefttmo* is effective only when the *tskstat* is TTS_WAI or TTS_WAS, and the *tskwait* is other than TTW_DLY.

  Note The *lefttmo* is undefined when the *tskwait* is TTW_DLY.

- *actcnt*
  Stores the activation request count.

- *wupcnt*
  Stores the wake-up request count.
  The *wupcnt* is effective only when the tskstat is other than TTS_DMT.

- *suscnt*
  Stores the suspension count.
  The *suscnt* is effective only when the tskstat is other than TTS_DMT.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *tskid* < 0 - *tskid* > VTMAX_TSK - When this service call was issued from a non-task, TSK_SELF was specified for *tskid*.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”. Note When the iref_tsk is issued from task or the ref_tsk is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.

ref_tst
iref_tst

Outline

Reference task state (simplified version).

C format

```
ER      ref_tst (ID tskid, T_RTST *pk_rtst);
ER      iref_tst (ID tskid, T_RTST *pk_rtst);
```

Parameter(s)

I/O	Parameter	Description
I	`ID tskid;`	ID number of the task. TSK_SELF: Invoking task. Value: ID number of the task.
O	`T_RTST *pk_rtst;`	Pointer to the packet returning the task state.

[Task state packet (simplified version): T_RTST]

```
typedef struct  t_rtst {
    STAT    tskstat;        /*Current state*/
    STAT    tskwait;        /*Reason for waiting*/
} T_RTST;
```

Explanation

Stores task state packet (current state, reason for waiting) of the task specified by parameter *tskid* in the area specified by parameter *pk_rtst*.
Used for referencing only the current state and reason for wait among task information.
Response becomes faster than using ref_tsk or iref_tsk because only a few information items are acquired.

- *tskstat*
  Stores the current state.

  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

- *tskwait*
  Stores the reason for waiting.
  The *tskwait* is effective only when the *tskstat* is TTS_WAI or TTS_WAS.

  TTW_SLP: Sleeping state caused by slp_tsk or tslp_tsk
  TTW_DLY: Delayed state caused by dly_tsk

TTW_SEM:	WAITING state for a semaphore resource caused by wai_sem or twai_sem
TTW_FLG:	WAITING state for an eventflag caused by wai_flg or twai_flg
TTW_SDTQ:	Sending WAITING state for a data queue caused by snd_dtq or tsnd_dtq
TTW_RDTQ:	Receiving WAITING state for a data queue caused by rcv_dtq or trcv_dtq
TTW_MBX:	Receiving WAITING state for a mailbox caused by rcv_mbx or trcv_mbx
TTW_MTX:	WAITING state for a mutex caused by loc_mtx or tloc_mtx
TTW_SMBF:	Sending WAITING state for a message buffer caused by snd_mbf or tsnd_mbf
TTW_RMBF:	Receiving WAITING state for a message buffer caused by rcv_mbf or trcv_mbf
TTW_MPF:	WAITING state for a fixed-sized memory block caused by get_mpf or tget_mpf
TTW_MPL:	WAITING state for a variable-sized memory block caused by get_mpl or tget_mpl

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *tskid* < 0 - *tskid* > VTMAX_TSK - When this service call was issued from a non-task, TSK_SELF was specified for *tskid*.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status "PSW.IPL > kernel interrupt mask level". Note When the iref_tst is issued from task or the ref_tst is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.

18.2.2 Task dependent synchronization functions

The following shows the service calls provided by the RI600V4 as the task dependent synchronization functions.

Table 18-4 Task Dependent Synchronization Functions

Service Call	Function	Useful Range
slp_tsk	Put task to sleep (waiting forever)	Task
tslp_tsk	Put task to sleep (with time-out)	Task
wup_tsk	Wake-up task	Task
iwup_tsk	Wake-up task	Non-task
can_wup	Cancel task wake-up requests	Task
ican_wup	Cancel task wake-up requests	Non-task
rel_wai	Release task from waiting	Task
irel_wai	Release task from waiting	Non-task
sus_tsk	Suspend task	Task
isus_tsk	Suspend task	Non-task
rsm_tsk	Resume suspended task	Task
irsm_tsk	Resume suspended task	Non-task
frsm_tsk	Forcibly resume suspended task	Task
ifrsm_tsk	Forcibly resume suspended task	Non-task
dly_tsk	Delay task	Task

slp_tsk

Outline

Put task to sleep (waiting forever).

C format

```
ER      slp_tsk (void);
```

Parameter(s)

None.

Explanation

As a result, the invoking task is unlinked from the ready queue and excluded from the RI600V4 scheduling subject.
If a wake-up request has been queued to the target task (the wake-up request counter > 0) when this service call is issued, this service call does not move the state but decrements the wake-up request counter (by subtracting 1 from the wake-up request counter).
The sleeping state is cancelled in the following cases.

Sleeping State Cancel Operation	Return Value
A wake-up request was issued as a result of issuing wup_tsk.	E_OK
A wake-up 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_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.

tslp_tsk

Outline

Put task to sleep (with time-out).

C format

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

Parameter(s)

I/O	Parameter	Description
I	TMO tmout;	Specified time-out (in millisecond). TMO_FEVR: Waiting forever. TMO_POL: Polling. Value: Specified time-out.

Explanation

This service call moves the 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 RI600V4 scheduling subject.
If a wake-up request has been queued to the target task (the wake-up request counter > 0) when this service call is issued, this service call does not move the state but decrements the wake-up request counter (by subtracting 1 from the wake-up request counter).
The sleeping state is cancelled in the following cases.

Sleeping State Cancel Operation	Return Value
A wake-up request was issued as a result of issuing wup_tsk.	E_OK
A wake-up 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 time specified by *tmout* has elapsed.	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.

Macro	Value	Description
E_PAR	-17	Parameter error. - *tmout* < -1 - *tmout* > (0x7FFFFFFF - TIC_NUME) / TIC_DENO
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
E_TMOUT	-50	Polling failure or specified time has elapsed.

wup_tsk
iwup_tsk

Outline

Wake-up task.

C format

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

Parameter(s)

I/O	Parameter	Description
I	`ID tskid;`	ID number of the task. TSK_SELF: Invoking task. Value: ID number of the task.

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 wake-up request counter (by added 1 to the wake-up request counter).

Note The wake-up request counter managed by the RI600V4 is configured in 8-bit widths. If the number of wake-up requests exceeds the maximum count value 255 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_ID	-18	Invalid ID number. - *tskid* < 0 - *tskid* > VTMAX_TSK - When iwup_tsk was issued from a non-task, TSK_SELF was specified for *tskid*.

Macro	Value	Description
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - The iwup_tsk was issued from task. - The wup_tsk was issued from non-task. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_OBJ	-41	Object state error. - Specified task is in the DORMANT state.
E_QOVR	-43	Queue overflow. - Wake-up request count exceeded 255.

can_wup
ican_wup

Outline

Cancel task wake-up requests.

C format

```
ER_UINT can_wup (ID tskid);
ER_UINT ican_wup (ID tskid);
```

Parameter(s)

I/O	Parameter	Description
I	`ID tskid;`	ID number of the task. TSK_SELF: Invoking task. Value: ID number of the task.

Explanation

These service calls cancel all of the wake-up requests queued to the task specified by parameter *tskid* (the wake-up request counter is set to 0), and return the number of cancelled wake-up requests.

Return value

Macro	Value	Description
E_ID	-18	Invalid ID number. - *tskid* < 0 - *tskid* > VTMAX_TSK - When this service call was issued from a non-task, TSK_SELF was specified for *tskid*.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”. Note When the ican_wup is issued from task or the can_wup is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.
E_OBJ	-41	Object state error. - Specified task is in the DORMANT state.
-	0 or more	Normal completion (wake-up request count).

rel_wai
irel_wai

Outline

Release task from waiting.

C format

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

Parameter(s)

I/O	Parameter	Description
I	ID *tskid*;	ID number of the task.

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 These service calls do not perform queuing of forced cancelation requests. If the target task is neither in the WAITING state nor WAITING-SUSPENDED state, "E_OBJ" is returned.

Note 2 The SUSPENDED state is not cancelled by these service calls.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *tskid* ≤ 0 - *tskid* > VTMAX_TSK
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - The irel_wai was issued from task. - The rel_wai was issued from non-task. - This service call was issued in the status "PSW.IPL > kernel interrupt mask level".

Macro	Value	Description
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);
```

Parameter(s)

I/O	Parameter	Description
I	ID tskid;	ID number of the task. TSK_SELF: Invoking task. Value: ID number of the task.

Explanation

These service calls move the task specified by parameter *tskid* 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, these service calls return "E_QOVR".

Note In the RI600V4, the suspend request can not be nested.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *tskid* < 0 - *tskid* > VTMAX_TSK - When this service call was issued from a non-task, TSK_SELF was specified for *tskid*.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - The isus_tsk was issued from task. - The sus_tsk was issued from non-task. - This service call was issued in the status "PSW.IPL > kernel interrupt mask level". - The invoking task is specified in the dispatching disabled state.

Macro	Value	Description
E_OBJ	-41	Object state error. - Specified task is in the DORMANT state. - Specified task is in the RUNNING state when isus_tsk is issued in the dispatching disabled state.
E_QOVR	-43	Queue overflow. - Specified task is neither in the SUSPENDED state nor WAITING-SUSPENDED state.

rsm_tsk
irsm_tsk

Outline

Resume suspended task.

C format

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

Parameter(s)

I/O	Parameter	Description
I	`ID tskid;`	ID number of the task.

Explanation

These service calls move the task specified by parameter *tskid* from the SUSPENDED state to the READY state, or from the WAITING-SUSPENDED state to the WAITING state.

Note 1 These service calls do not perform queuing of forced cancelation requests. If the target task is neither in the SUSPENDED state nor WAITING-SUSPENDED state, “E_OBJ” is returned.

Note 2 The RI600V4 does not support queuing of suspend request. The behavior of the frsm_tsk and ifrsm_tsk, that can release from the SUSPENDED state even if suspend request has been queued, are same as rsm_tsk and irsm_tsk.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *tskid* ≤ 0 - *tskid* > VTMAX_TSK
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - The irsm_tsk was issued from task. - The rsm_tsk was issued from non-task. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.

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);
```

Parameter(s)

I/O	Parameter	Description
I	`ID tskid;`	ID number of the task.

Explanation

These service calls cancel all of the suspend requests issued for the task specified by parameter *tskid* (by setting the suspend request counter to 0). As a result, the target task moves from the SUSPENDED state to the READY state, or from the WAITING-SUSPENDED state to the WAITING state.

Note 1 These service calls do not perform queuing of forced cancelation requests. If the target task is neither in the SUSPENDED state nor WAITING-SUSPENDED state, “E_OBJ” is returned.

Note 2 The RI600V4 does not support queuing of suspend request. Therefore, the behavior of these service calls are same as rsm_tsk and irsm_tsk.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *tskid* ≤ 0 - *tskid* > VTMAX_TSK
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - The ifrsm_tsk was issued from task. - The frsm_tsk was issued from non-task. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.

Macro	Value	Description
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);
```

Parameter(s)

I/O	Parameter	Description
I	`RELTIM dlytim;`	Amount of time to delay the invoking task (in millisecond).

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 RI600V4 scheduling subject.
The delayed state is cancelled in the following cases.

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

Note When 0 is specified as *dlytim*, the delay time is up to next base clock interrupt generation.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *dlytim* > (0x7FFFFFFF - TIC_NUME) / TIC_DENO
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.

Macro	Value	Description
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.

18.2.3 Synchronization and communication functions (semaphores)

The following shows the service calls provided by the RI600V4 as the synchronization and communication functions (semaphores).

Table 18-5 Synchronization and Communication Functions (Semaphores)

Service Call	Function	Useful Range
wai_sem	Acquire semaphore resource (waiting forever)	Task
pol_sem	Acquire semaphore resource (polling)	Task
ipol_sem	Acquire semaphore resource (polling)	Non-task
twai_sem	Acquire semaphore resource (with time-out)	Task
sig_sem	Release semaphore resource	Task
isig_sem	Release semaphore resource	Non-task
ref_sem	Reference semaphore state	Task
iref_sem	Reference semaphore state	Non-task

wai_sem

Outline

Acquire semaphore resource (waiting forever).

C format

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

Parameter(s)

I/O	Parameter	Description
I	`ID semid;`	ID number of the semaphore.

Explanation

This service call acquires a resource from the semaphore specified by parameter *semid* (subtracts 1 from the semaphore counter).
If no resources are acquired from the target semaphore when this service call is issued (no available resources exist), this service call does not acquire resources but queues the invoking task to the target semaphore wait queue and moves it from the RUNNING state to the WAITING state (resource acquisition wait state).
The WAITING state for a semaphore resource is cancelled in the following cases.

WAITING State for a Semaphore Resource Cancel Operation	Return Value
The resource was released to the target semaphore as a result of issuing sig_sem.	E_OK
The resource was released 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

Note Invoking tasks are queued to the target semaphore wait queue in the order defined during configuration (FIFO order or current priority order).

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *semid* ≤ 0 - *semid* > VTMAX_SEM

Macro	Value	Description
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.

pol_sem
ipol_sem

Outline

Acquire semaphore resource (polling).

C format

```
ER      pol_sem (ID semid);
ER      isem_sem (ID semid);
```

Parameter(s)

I/O	Parameter	Description
I	`ID semid;`	ID number of the semaphore.

Explanation

This service call acquires a resource from the semaphore specified by parameter *semid* (subtracts 1 from the semaphore counter).
If a resource could not be acquired from the target semaphore (semaphore counter is set to 0) 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_ID	-18	Invalid ID number. - *semid* ≤ 0 - *semid* > VTMAX_SEM
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”. Note When the ipol_sem is issued from task or the pol_sem is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.
E_TMOUT	-50	Polling failure.

twai_sem

Outline

Acquire semaphore resource (with time-out).

C format

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

Parameter(s)

I/O	Parameter	Description
I	`ID semid;`	ID number of the semaphore.
I	`TMO tmout;`	Specified time-out (in millisecond). TMO_FEVR: Waiting forever. TMO_POL: Polling. Value: Specified time-out.

Explanation

This service call acquires a resource from the semaphore specified by parameter *semid* (subtracts 1 from the semaphore counter).

If no resources are acquired from the target semaphore when service call is issued this (no available resources exist), this service call does not acquire resources but queues the invoking task to the target semaphore wait queue and moves it from the RUNNING state to the WAITING state with time-out (resource acquisition wait state).

The WAITING state for a semaphore resource is cancelled in the following cases.

WAITING State for a Semaphore Resource Cancel Operation	Return Value
The resource was released to the target semaphore as a result of issuing sig_sem.	E_OK
The resource was released 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 time specified by *tmout* has elapsed.	E_TMOUT

Note 1 Invoking tasks are queued to the target semaphore wait queue in the order defined during configuration (FIFO order or current priority order).

Note 2 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_PAR	-17	Parameter error. - *tmout* < -1 - *tmout* > (0x7FFFFFFF - TIC_NUME) / TIC_DENO
E_ID	-18	Invalid ID number. - *semid* ≤ 0 - *semid* > VTMAX_SEM
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
E_TMOUT	-50	Polling failure or specified time has elapsed.

sig_sem
isig_sem

Outline

Release semaphore resource.

C format

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

Parameter(s)

I/O	Parameter	Description
I	`ID semid;`	ID number of the semaphore.

Explanation

These service calls releases the resource to the semaphore specified by parameter *semid* (adds 1 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 With the RI600V4, the maximum possible number of semaphore resources (Maximum resource count (max_count)) is defined during configuration. If the number of resources exceeds the specified maximum resource count, this service call therefore does not release the acquired resources (addition to the semaphore counter value) but returns E_QOVR.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *semid* ≤ 0 - *semid* > VTMAX_SEM
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - The isig_sem was issued from task. - The sig_sem was issued from non-task. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.

Macro	Value	Description
E_QOVR	-43	Queue overflow. - Resource count exceeded the Maximum resource count (max_count).

ref_sem
iref_sem

Outline

Reference semaphore state.

C format

```
ER      ref_sem (ID semid, T_RSEM *pk_rsem);
ER      iref_sem (ID semid, T_RSEM *pk_rsem);
```

Parameter(s)

I/O	Parameter	Description
I	`ID semid;`	ID number of the semaphore.
O	`T_RSEM *pk_rsem;`	Pointer to the packet returning the semaphore state.

[Semaphore state packet: T_RSEM]

```
typedef struct  t_rsem {
    ID      wtskid;         /*Existence of waiting task*/
    UINT    semcnt;         /*Current resource count*/
} T_RSEM;
```

Explanation

Stores semaphore state packet (ID number of the task at the head of the wait queue, current resource count, etc.) of the semaphore specified by parameter *semid* in the area specified by parameter *pk_rsem*.

- *wtskid*
  Stores whether a task is queued to the semaphore wait queue.

  TSK_NONE: No applicable task
  Value: ID number of the task at the head of the wait queue

- *semcnt*
  Stores the current resource count.

Return value

Macro	Value	Description
E_OK	0	Normal completion.

Macro	Value	Description
E_ID	-18	Invalid ID number. - *semid* $\leq$ 0 - *semid* > VTMAX_SEM
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”. Note When the iref_sem is issued from task or the ref_sem is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.

18.2.4 Synchronization and communication functions (eventflags)

The following shows the service calls provided by the RI600V4 as the synchronization and communication functions (eventflags).

Table 18-6 Synchronization and Communication Functions (Eventflags)

Service Call	Function	Useful Range
set_flg	Set eventflag	Task
iset_flg	Set eventflag	Non-task
clr_flg	Clear eventflag	Task
iclr_flg	Clear eventflag	Non-task
wai_flg	Wait for eventflag (waiting forever)	Task
pol_flg	Wait for eventflag (polling)	Task
ipol_flg	Wait for eventflag (polling)	Non-task
twai_flg	Wait for eventflag (with time-out)	Task
ref_flg	Reference eventflag state	Task
iref_flg	Reference eventflag state	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);
```

Parameter(s)

I/O	Parameter	Description
I	`ID` *`flgid;`*	ID number of the eventflag.
I	`FLGPTN` *`setptn;`*	Bit pattern to set.

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.
After that, these service calls evaluate whether the wait condition of the tasks in the wait queue is satisfied. This evaluation is done in order of the wait queue. If the wait condition is satisfied, 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. At this time, the bit pattern of the target event flag is cleared to 0 and this service call finishes processing if the TA_CLR attribute is specified for the target eventflag.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *flgid* ≤ 0 - *flgid* > VTMAX_FLG
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - The iset_flg was issued from task. - The set_flg was issued from non-task. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.

clr_flg
iclr_flg

Outline

Clear eventflag.

C format

```
ER      clr_flg (ID flgid, FLGPTN clrptn);
ER      iclr_flg (ID flgid, FLGPTN clrptn);
```

Parameter(s)

I/O	Parameter	Description
I	`ID flgid;`	ID number of the eventflag.
I	`FLGPTN clrptn;`	Bit pattern to clear.

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.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *flgid* $\leq$ 0 - *flgid* > VTMAX_FLG
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status "PSW.IPL > kernel interrupt mask level". Note When the iclr_flg is issued from task or the clr_flg is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.

wai_flg

Outline

Wait for eventflag (waiting forever).

C format

```
ER      wai_flg (ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn);
```

Parameter(s)

I/O	Parameter	Description
I	`ID flgid;`	ID number of the eventflag.
I	`FLGPTN waiptn;`	Wait bit pattern.
I	`MODE wfmode;`	Wait mode. TWF_ANDW: AND waiting condition. 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).
The WAITING state for an eventflag is cancelled in the following cases.

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.

Note 1 With the RI600V4, whether to enable queuing of multiple tasks to the event flag wait queue is defined during configuration. If this service call is issued for the event flag (TA_WSGL attribute) to which a wait task is queued, therefore, “E_ILUSE” is returned regardless of whether the required condition is immediately satisfied.

TA_WSGL: Only one task is allowed to be in the WAITING state for the eventflag.
TA_WMUL: Multiple tasks are allowed to be in the WAITING state for the eventflag.

Note 2 Invoking tasks are queued to the target event flag (TA_WMUL attribute) wait queue in the order defined during configuration (FIFO order or current priority order).
However, when the TA_CLR attribute is not specified, the wait queue is managed in the FIFO order even if the priority order is specified. This behavior falls outside μITRON4.0 specification.

Note 3 The RI600V4 performs bit pattern clear processing (0 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_PAR	-17	Parameter error. - *waiptn* == 0 - *wfmode* is invalid.
E_ID	-18	Invalid ID number. - *flgid* ≤ 0 - *flgid* > VTMAX_FLG
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_ILUSE	-28	Illegal use of service call. - There is already a task waiting for an eventflag with the TA_WSGL attribute.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.

pol_flg
ipol_flg

Outline

Wait for eventflag (polling).

C format

```
ER      pol_flg (ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn);
ER      ipol_flg (ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn);
```

Parameter(s)

I/O	Parameter	Description
I	`ID flgid;`	ID number of the eventflag.
I	`FLGPTN waiptn;`	Wait bit pattern.
I	`MODE wfmode;`	Wait mode. TWF_ANDW: AND waiting condition. 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 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 With the RI600V4, whether to enable queuing of multiple tasks to the event flag wait queue is defined during configuration. If this service call is issued for the event flag (TA_WSGL attribute) to which a wait task is queued, therefore, “E_ILUSE” is returned regardless of whether the required condition is immediately satisfied.

TA_WSGL: Only one task is allowed to be in the WAITING state for the eventflag.
TA_WMUL: Multiple tasks are allowed to be in the WAITING state for the eventflag.

Note 2 The RI600V4 performs bit pattern clear processing (0 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_PAR	-17	Parameter error. - *waiptn* == 0 - *wfmode* is invalid.
E_ID	-18	Invalid ID number. - *flgid* $\leq$ 0 - *flgid* > VTMAX_FLG
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”. Note When the ipol_flg is issued from task or the pol_flg is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.
E_ILUSE	-28	Illegal use of service call. - 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 time-out).

C format

```
ER      twai_flg (ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn, TMO tmout);
```

Parameter(s)

I/O	Parameter	Description
I	`ID flgid;`	ID number of the eventflag.
I	`FLGPTN waiptn;`	Wait bit pattern.
I	`MODE wfmode;`	Wait mode. TWF_ANDW: AND waiting condition. TWF_ORW: OR waiting condition.
O	`FLGPTN *p_flgptn;`	Bit pattern causing a task to be released from waiting.
I	`TMO tmout;`	Specified time-out (in millisecond). TMO_FEVR: Waiting forever. TMO_POL: Polling. Value: Specified time-out.

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).
The WAITING state for an eventflag is cancelled in the following cases.

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 time specified by *tmout* has elapsed.	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 With the RI600V4, whether to enable queuing of multiple tasks to the event flag wait queue is defined during configuration. If this service call is issued for the event flag (TA_WSGL attribute) to which a wait task is queued, therefore, “E_ILUSE” is returned regardless of whether the required condition is immediately satisfied.

TA_WSGL: Only one task is allowed to be in the WAITING state for the eventflag.
TA_WMUL: Multiple tasks are allowed to be in the WAITING state for the eventflag.

Note 2 Invoking tasks are queued to the target event flag (TA_WMUL attribute) wait queue in the order defined during configuration (FIFO order or current priority order).
However, when the TA_CLR attribute is not specified, the wait queue is managed in the FIFO order even if the priority order is specified. This behavior falls outside μITRON4.0 specification.

Note 3 The RI600V4 performs bit pattern clear processing (0 setting) when the required condition of the target eventflag (TA_CLR attribute) is satisfied.

Note 4 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_PAR	-17	Parameter error. - *waiptn* == 0 - *wfmode* is invalid. - *tmout* < -1 - *tmout* > (0x7FFFFFFF - TIC_NUME) / TIC_DENO
E_ID	-18	Invalid ID number. - *flgid* ≤ 0 - *flgid* > VTMAX_FLG
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_ILUSE	-28	Illegal use of service call. - There is already a task waiting for an eventflag with the TA_WSGL attribute.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
E_TMOUT	-50	Polling failure or specified time has elapsed.

ref_flg
iref_flg

Outline

Reference eventflag state.

C format

```
ER      ref_flg (ID flgid, T_RFLG *pk_rflg);
ER      iref_flg (ID flgid, T_RFLG *pk_rflg);
```

Parameter(s)

I/O	Parameter	Description
I	`ID flgid;`	ID number of the eventflag.
O	`T_RFLG *pk_rflg;`	Pointer to the packet returning the eventflag state.

[Eventflag state packet: T_RFLG]

```
typedef struct  t_rflg {
    ID      wtskid;         /*Existence of waiting task*/
    FLGPTN  flgptn;         /*Current bit pattern*/
} T_RFLG;
```

Explanation

Stores eventflag state packet (ID number of the task at the head of the wait queue, current bit pattern, etc.) of the eventflag specified by parameter *flgid* in the area specified by parameter *pk_rflg*.

- *wtskid*
  Stores whether a task is queued to the event flag wait queue.

  TSK_NONE: No applicable task
  Value: ID number of the task at the head of the wait queue

- *flgptn*
  Stores the current bit pattern.

Return value

Macro	Value	Description
E_OK	0	Normal completion.

Macro	Value	Description
E_ID	-18	Invalid ID number. - *flgid* ≤ 0 - *flgid* > VTMAX_FLG
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”. Note When the iref_flg is issued from task or the ref_flg is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.

18.2.5 Synchronization and communication functions (data queues)

The following shows the service calls provided by the RI600V4 as the synchronization and communication functions (data queues).

Table 18-7 Synchronization and Communication Functions (Data Queues)

Service Call	Function	Useful Range
snd_dtq	Send to data queue (waiting forever)	Task
psnd_dtq	Send to data queue (polling)	Task
ipsnd_dtq	Send to data queue (polling)	Non-task
tsnd_dtq	Send to data queue (with time-out)	Task
fsnd_dtq	Forced send to data queue	Task
ifsnd_dtq	Forced send to data queue	Non-task
rcv_dtq	Receive from data queue (waiting forever)	Task
prcv_dtq	Receive from data queue (polling)	Task
iprcv_dtq	Receive from data queue (polling)	Non-task
trcv_dtq	Receive from data queue (with time-out)	Task
ref_dtq	Reference data queue state	Task
iref_dtq	Reference data queue state	Non-task

snd_dtq

Outline

Send to data queue (waiting forever).

C format

```
ER      snd_dtq (ID dtqid, VP_INT data);
```

Parameter(s)

I/O	Parameter	Description
I	`ID dtqid;`	ID number of the data queue.
I	`VP_INT data;`	Data element to be sent to the data queue.

Explanation

This service call processes as follows according to the situation of the data queue specified by the parameter *dtqid*.

- There is a task in the reception wait queue.
  This service call transfers the data specified by parameter *data* to the task in the top of the reception wait queue. 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.

- There is no task neither in the reception wait queue and transmission wait queue and there is available space in the data queue.
  This service call stores the data specified by parameter *data* to the data queue.

- There is no task neither in the reception wait queue and transmission wait queue and there is no available space in the data queue, or there is a task in the transmission wait queue.
  This service call 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.

Sending WAITING State for a Data Queue Cancel Operation	Return Value
Available space was secured in the data queue area as a result of issuing rcv_dtq.	E_OK
Available space was secured in the data queue area as a result of issuing prcv_dtq.	E_OK
Available space was secured in the data queue area as a result of issuing iprcv_dtq.	E_OK
Available space was secured in the data queue area 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
The data queue is reset as a result of issuingissuing vrst_dtq.	EV_RST

Note 1 Data is written to the data queue area 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 or current priority order).

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *dtqid* ≤ 0 - *dtqid* > VTMAX_DTQ
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
EV_RST	-127	Released from WAITING state by the object reset (vrst_dtq)

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);
```

Parameter(s)

I/O	Parameter	Description
I	`ID dtqid;`	ID number of the data queue.
I	`VP_INT data;`	Data element to be sent to the data queue.

Explanation

These service calls process as follows according to the situation of the data queue specified by the parameter *dtqid*.

- There is a task in the reception wait queue.
  These service calls transfer the data specified by parameter *data* to the task in the top of the reception wait queue. 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.
- There is no task neither in the reception wait queue and transmission wait queue and there is available space in the data queue.
  These service calls store the data specified by parameter *data* to the data queue.
- There is no task neither in the reception wait queue and transmission wait queue and there is no available space in the data queue, or there is a task in the transmission wait queue.
  These service calls return “E_TMOUT”.

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_ID	-18	Invalid ID number. - *dtqid* ≤ 0 - *dtqid* > VTMAX_DTQ

Macro	Value	Description
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - The ipsnd_dtq was issued from task. - The psnd_dtq was issued from non-task. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_TMOUT	-50	Polling failure.

tsnd_dtq

Outline

Send to data queue (with time-out).

C format

```
ER      tsnd_dtq (ID dtqid, VP_INT data, TMO tmout);
```

Parameter(s)

I/O	Parameter	Description
I	`ID dtqid;`	ID number of the data queue.
I	`VP_INT data;`	Data element to be sent to the data queue.
I	`TMO tmout;`	Specified time-out (in millisecond). TMO_FEVR: Waiting forever. TMO_POL: Polling. Value: Specified time-out.

Explanation

This service call processes as follows according to the situation of the data queue specified by the parameter *dtqid*.

- There is a task in the reception wait queue.
  This service call transfers the data specified by parameter *data* to the task in the top of the reception wait queue. 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.
- There is no task neither in the reception wait queue and transmission wait queue and there is available space in the data queue.
  This service call stores the data specified by parameter *data* to the data queue.
- There is no task neither in the reception wait queue and transmission wait queue and there is no available space in the data queue, or there is a task in the transmission wait queue.
  This service call 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.

Sending WAITING State for a Data Queue Cancel Operation	Return Value
Available space was secured in the data queue area as a result of issuing rcv_dtq.	E_OK
Available space was secured in the data queue area as a result of issuing prcv_dtq.	E_OK
Available space was secured in the data queue area as a result of issuing iprcv_dtq.	E_OK
Available space was secured in the data queue area 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
The data queue is reset as a result of issuing vrst_dtq.	EV_RST

Sending WAITING State for a Data Queue Cancel Operation	Return Value
The time specified by *tmout* has elapsed.	E_TMOUT

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 or current priority 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 will be executed.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *tmout* < -1 - *tmout* > (0x7FFFFFFF - TIC_NUME) / TIC_DENO
E_ID	-18	Invalid ID number. - *dtqid* ≤ 0 - *dtqid* > VTMAX_DTQ
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
E_TMOUT	-50	Polling failure or specified time has elapsed.
EV_RST	-127	Released from WAITING state by the object reset (vrst_dtq)

fsnd_dtq
ifsnd_dtq

Outline

Forced send to data queue.

C format

```
ER      fsnd_dtq (ID dtqid, VP_INT data);
ER      ifsnd_dtq (ID dtqid, VP_INT data);
```

Parameter(s)

I/O	Parameter	Description
I	`ID dtqid;`	ID number of the data queue.
I	`VP_INT data;`	Data element to be sent to the data queue.

Explanation

These service calls process as follows according to the situation of the data queue specified by the parameter *dtqid*.

- There is a task in the reception wait queue.
  This service call transfers the data specified by parameter *data* to the task in the top of the reception wait queue. 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.
- There is no task neither in the reception wait queue and transmission wait queue.
  This service call stores the data specified by parameter *data* to the data queue.
  If there is no available space in the data queue, this service call deletes the oldest data in the data queue before storing the data specified by *data* to the data queue.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *dtqid* ≤ 0 - *dtqid* > VTMAX_DTQ

Macro	Value	Description
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - The ifsnd_dtq was issued from task. - The fsnd_dtq was issued from non-task. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_ILUSE	-28	Illegal use of service call. - The capacity of the data queue area is 0.

rcv_dtq

Outline

Receive from data queue (waiting forever).

C format

```
ER      rcv_dtq (ID dtqid, VP_INT *p_data);
```

Parameter(s)

I/O	Parameter	Description
I	`ID dtqid;`	ID number of the data queue.
O	`VP_INT *p_data;`	Data element received from the data queue.

Explanation

This service call processes as follows according to the situation of the data queue specified by the parameter *dtqid*.

- There is a data in the data queue.
  This service call takes out the oldest data from the data queue and stores the data to the area specified by *p_data*.
  When there is a task in the transmission wait queue, this service call stores the data sent by the task in the top of the transmission wait queue and moves it from the WAITING state (data transmission wait state) to the READY state.

- There is no data in the data queue and there is a task in the transmission wait queue.
  This service call stores the data specified by the task in the top of the transmission wait queue to the area specified by *p_data*. As a result, the task is unlinked from the transmission wait queue and moves from the WAITING state (data transmission wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  Note, this situation is caused only when the capacity of the data queue is 0.

- There is no data in the data queue and there is no task in the transmission wait queue.
  This service call 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.

Receiving WAITING State for a Data Queue Cancel Operation	Return Value
Data was sent to the data queue area as a result of issuing snd_dtq.	E_OK
Data was sent to the data queue area as a result of issuing psnd_dtq.	E_OK
Data was sent to the data queue area as a result of issuing ipsnd_dtq.	E_OK
Data was sent to the data queue area as a result of issuing tsnd_dtq.	E_OK
Data was sent to the data queue area as a result of issuing fsnd_dtq.	E_OK
Data was sent to the data queue area 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 Invoking tasks are queued to the reception wait queue of the target data queue in the order of the data reception request.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *dtqid* ≤ 0 - *dtqid* > VTMAX_DTQ
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.

prcv_dtq
iprcv_dtq

Outline

Receive from data queue (polling).

C format

```
ER      prcv_dtq (ID dtqid, VP_INT *p_data);
ER      iprcv_dtq (ID dtqid, VP_INT *p_data);
```

Parameter(s)

I/O	Parameter	Description
I	`ID dtqid;`	ID number of the data queue.
O	`VP_INT *p_data;`	Data element received from the data queue.

Explanation

These service calls process as follows according to the situation of the data queue specified by the parameter *dtqid*.

- There is a data in the data queue.
  This service call takes out the oldest data from the data queue and stores the data to the area specified by *p_data*.
  When there is a task in the transmission wait queue, this service call stores the data sent by the task in the top of the transmission wait queue and moves it from the WAITING state (data transmission wait state) to the READY state.
- There is no data in the data queue and there is a task in the transmission wait queue.
  These service calls store the data specified by the task in the top of the transmission wait queue to the area specified by *p_data*. As a result, the task is unlinked from the transmission wait queue and moves from the WAITING state (data transmission wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  Note, this situation is caused only when the capacity of the data queue is 0.
- There is no data in the data queue and there is no task in the transmission wait queue.
  These service calls return "E_TMOUT".

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *dtqid* ≤ 0 - *dtqid* > VTMAX_DTQ

Macro	Value	Description
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - The iprcv_dtq was issued from task. - The prcv_dtq was issued from non-task. - This service call was issued in the status "PSW.IPL > kernel interrupt mask level".
E_TMOUT	-50	Polling failure.

trcv_dtq

Outline

Receive from data queue (with time-out).

C format

```
ER      trcv_dtq (ID dtqid, VP_INT *p_data, TMO tmout);
```

Parameter(s)

I/O	Parameter	Description
I	`ID dtqid;`	ID number of the data queue.
O	`VP_INT *p_data;`	Data element received from the data queue.
I	`TMO tmout;`	Specified time-out (in millisecond). TMO_FEVR: Waiting forever. TMO_POL: Polling. Value: Specified time-out.

Explanation

This service call processes as follows according to the situation of the data queue specified by the parameter *dtqid*.

- There is a data in the data queue.
  This service call takes out the oldest data from the data queue and stores the data to the area specified by *p_data*.
  When there is a task in the transmission wait queue, this service call stores the data sent by the task in the top of the transmission wait queue and moves it from the WAITING state (data transmission wait state) to the READY state.

- There is no data in the data queue and there is a task in the transmission wait queue.
  This service call stores the data specified by the task in the top of the transmission wait queue to the area specified by *p_data*. As a result, the task is unlinked from the transmission wait queue and moves from the WAITING state (data transmission wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  Note, this situation is caused only when the capacity of the data queue is 0.

- There is no data in the data queue and there is no task in the transmission wait queue.
  This service call 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 (data reception wait state).
  The receiving WAITING state for a data queue is cancelled in the following cases.

Receiving WAITING State for a Data Queue Cancel Operation	Return Value
Data was sent to the data queue area as a result of issuing snd_dtq.	E_OK
Data was sent to the data queue area as a result of issuing psnd_dtq.	E_OK
Data was sent to the data queue area as a result of issuing ipsnd_dtq.	E_OK
Data was sent to the data queue area as a result of issuing tsnd_dtq.	E_OK
Data was sent to the data queue area as a result of issuing fsnd_dtq.	E_OK
Data was sent to the data queue area as a result of issuing ifsnd_dtq.	E_OK

Receiving WAITING State for a Data Queue Cancel Operation	Return Value
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI
The time specified by *tmout* has elapsed.	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 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_PAR	-17	Parameter error. - *tmout* < -1 - *tmout* > (0x7FFFFFFF - TIC_NUME) / TIC_DENO
E_ID	-18	Invalid ID number. - *dtqid* ≤ 0 - *dtqid* > VTMAX_DTQ
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state. - This service call was issued in the status "PSW.IPL > kernel interrupt mask level".
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
E_TMOUT	-50	Polling failure or specified time has elapsed.

ref_dtq
iref_dtq

Outline

Reference data queue state.

C format

```
ER      ref_dtq (ID dtqid, T_RDTQ *pk_rdtq);
ER      iref_dtq (ID dtqid, T_RDTQ *pk_rdtq);
```

Parameter(s)

I/O	Parameter	Description
I	`ID dtqid;`	ID number of the data queue.
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*.

- *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 transmission 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 reception wait queue

- *sdtqcnt*
  Stores the number of data elements in data queue.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *dtqid* ≤ 0 - *dtqid* > VTMAX_DTQ
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”. Note When the iref_dtq is issued from task or the ref_dtq is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.

18.2.6 Synchronization and communication functions (mailboxes)

The following shows the service calls provided by the RI600V4 as the synchronization and communication functions (mailboxes).

Table 18-8 Synchronization and Communication Functions (Mailboxes)

Service Call	Function	Useful Range
snd_mbx	Send to mailbox	Task
isnd_mbx	Send to mailbox	Non-task
rcv_mbx	Receive from mailbox (waiting forever)	Task
prcv_mbx	Receive from mailbox (polling)	Task
iprcv_mbx	Receive from mailbox (polling)	Non-task
trcv_mbx	Receive from mailbox (with time-out)	Task
ref_mbx	Reference mailbox state	Task
iref_mbx	Reference mailbox state	Non-task

snd_mbx
isnd_mbx

Outline

Send to mailbox.

C format

```
ER      snd_mbx (ID mbxid, T_MSG *pk_msg);
ER      isnd_mbx (ID mbxid, T_MSG *pk_msg);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mbxid;`	ID number of the mailbox.
I	`T_MSG *pk_msg;`	Start address of the message packet to be sent to the mailbox.

[Message packet T_MSG for TA_MFIFO attribute]

```
typedef struct {
    VP      msghead;        /*RI600V4 management area*/
} T_MSG;
```

[Message packet for T_MSG_PRI for TA_MPRI attribute]

```
typedef struct {
    T_MSG   msgque;         /*Message header*/
    PRI     msgpri;         /*Message priority*/
} T_MSG_PRI;
```

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 state for a mailbox) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

Note 1 Messages are queued to the target mailbox message queue in the order defined by queuing method during configuration (FIFO order or message priority order).

Note 2 Do not modify transmitted message (the area indicated by *pk_msg*) until the message is received.

Return value

Macro	Value	Description
E_OK	0	Normal completion.

E_PAR	-17	Parameter error.
		- When the target mailbox has TA_MPRI attribute:
		  - *msgpri* ≤ 0
		  - *msgpri* > TMAX_MPRI

E_ID	-18	Invalid ID number. - *mbxid* ≤ 0 - *mbxid* > VTMAX_MBX
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - The isnd_mbx was issued from task. - The snd_mbx was issued from non-task. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.

rcv_mbx

Outline

Receive from mailbox (waiting forever).

C format

```
ER      rcv_mbx (ID mbxid, T_MSG **ppk_msg);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mbxid;`	ID number of the mailbox.
O	`T_MSG **ppk_msg;`	Start address of the message packet received from the mailbox.

[Message packet T_MSG for TA_MFIFO attribute]

```
typedef struct {
    VP      msghead;        /*RI600V4 management area*/
} T_MSG;
```

[Message packet T_MSG_PRI for TA_MPRI attribute]

```
typedef struct {
    T_MSG   msgque;         /*Message header*/
    PRI     msgpri;         /*Message priority*/
} T_MSG_PRI;
```

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 no message could be received from the target mailbox (no messages were queued to the wait queue) when this service call is issued, this service call does not receive messages but queues the invoking task to the target mailbox wait queue and moves it from the RUNNING state to the WAITING state (message reception wait state).
The receiving WAITING state for a mailbox is cancelled in the following cases.

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
A message was transmitted to the target mailbox as a result of issuing isnd_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 Invoking tasks are queued to the target mailbox wait queue in the order defined during configuration (FIFO order or current priority order).

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *mbxid* $\le$ 0 - *mbxid* > VTMAX_MBX
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.

prcv_mbx
iprcv_mbx

Outline

Receive from mailbox (polling).

C format

```
ER      prcv_mbx (ID mbxid, T_MSG **ppk_msg);
ER      iprcv_mbx (ID mbxid, T_MSG **ppk_msg);
```

Parameter(s)

I/O	Parameter	Description
I	ID *mbxid*;	ID number of the mailbox.
O	T_MSG ****ppk_msg*;	Start address of the message packet received from the mailbox.

[M[Message packet T_MSG for TA_MFIFO attribute]

```
typedef struct {
    VP      msghead;       /*RI600V4 management area*/
} T_MSG;
```

[Message packet T_MSG_PRI for TA_MPRI attribute]

```
typedef struct {
    T_MSG   msgque;        /*Message header*/
    PRI     msgpri;        /*Message priority*/
} T_MSG_PRI;
```

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.

Return value

Macro	Value	Description
E_OK	0	Normal completion.

Macro	Value	Description
E_ID	-18	Invalid ID number. - *mbxid* ≤ 0 - *mbxid* > VTMAX_MBX
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”. Note When the iprcv_mbx is issued from task or the prcv_mbx is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.
E_TMOUT	-50	Polling failure.

trcv_mbx

Outline

Receive from mailbox (with time-out).

C format

```
ER      trcv_mbx (ID mbxid, T_MSG **ppk_msg, TMO tmout);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mbxid;`	ID number of the mailbox.
O	`T_MSG **ppk_msg;`	Start address of the message packet received from the mailbox.
I	`TMO tmout;`	Specified time-out (in millisecond). TMO_FEVR: Waiting forever. TMO_POL: Polling. Value: Specified time-out.

[Message packet: T_MSG]

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

[Message packet: T_MSG_PRI]

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

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 no message could be received from the target mailbox (no messages were queued to the wait queue) when this service call is issued, this service call does not receive messages but queues the invoking task to the target mailbox wait queue and moves it from the RUNNING state to the WAITING state with time-out (message reception wait state).
The receiving WAITING state for a mailbox is cancelled in the following cases.

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
A message was transmitted to the target mailbox as a result of issuing isnd_mbx.	E_OK

Receiving WAITING State for a Mailbox Cancel Operation	Return Value
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI
The time specified by *tmout* has elapsed.	E_TMOUT

Note 1 Invoking tasks are queued to the target mailbox wait queue in the order defined during configuration (FIFO order or current priority order).

Note 2 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.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *tmout* < -1 - *tmout* > (0x7FFFFFFF - TIC_NUME) / TIC_DENO
E_ID	-18	Invalid ID number. - *mbxid* ≤ 0 - *mbxid* > VTMAX_MBX
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
E_TMOUT	-50	Polling failure or specified time has elapsed.

ref_mbx
iref_mbx

Outline

Reference mailbox state.

C format

```
ER      ref_mbx (ID mbxid, T_RMBX *pk_rmbx);
ER      iref_mbx (ID mbxid, T_RMBX *pk_rmbx);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mbxid;`	ID number of the mailbox.
O	`T_RMBX *pk_rmbx;`	Pointer to the packet returning the mailbox state.

[Mailbox state packet: T_RMBX]

```
typedef struct  t_rmbx {
    ID      wtskid;         /*Existence of waiting task*/
    T_MSG   *pk_msg;        /*Existence of waiting message*/
} T_RMBX;
```

Explanation

Stores mailbox state packet (ID number of the task at the head of the wait queue, start address of the message packet at the head of the wait queue) of the mailbox specified by parameter *mbxid* in the area specified by parameter *pk_rmbx*.

- *wtskid*
  Stores whether a task is queued to the mailbox wait queue.

  TSK_NONE: No applicable task
  Value: ID number of the task at the head of the wait queue

- *pk_msg*
  Stores whether a message is queued to the mailbox wait queue.

  NULL: No applicable message
  Value: Start address of the message packet at the head of the wait queue

Return value

Macro	Value	Description
E_OK	0	Normal completion.

Macro	Value	Description
E_ID	-18	Invalid ID number. - *mbxid* ≤ 0 - *mbxid* > VTMAX_MBX
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”. Note When the iref_mbx is issued from task or the ref_mbx is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.

18.2.7 Extended synchronization and communication functions (mutexes)

The following shows the service calls provided by the RI600V4 as the extended synchronization and communication functions (mutexes).

Table 18-9 Extended Synchronization and Communication Functions (Mutexes)

Service Call	Function	Useful Range
loc_mtx	Lock mutex (waiting forever)	Task
ploc_mtx	Lock mutex (polling)	Task
tloc_mtx	Lock mutex (with time-out)	Task
unl_mtx	Unlock mutex	Task
ref_mtx	Reference mutex state	Task

loc_mtx

Outline

Lock mutex (waiting forever).

C format

```
ER       loc_mtx (ID mtxid);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mtxid;`	ID number of the mutex.

Explanation

This service call locks the mutex specified by parameter *mtxid*.
If the target mutex could not be locked (another task has been locked) when this service call is issued, this service call queues the invoking task to the target mutex wait queue and moves it from the RUNNING state to the WAITING state (mutex wait state).
The WAITING state for a mutex is cancelled in the following cases.

WAITING State for a Mutex Cancel Operation	Return Value
The locked state of the target mutex was cancelled as a result of issuing unl_mtx.	E_OK
The locked state of the target mutex was cancelled as a result of issuing ext_tsk.	E_OK
The locked state of the target mutex was cancelled as a result of issuing ter_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

When the mutex is locked, this service call changes the current priority of the invoking task to the ceiling priority of the target mutex. However, this service call does not change the current priority when the invoking task has locked other mutexes and the ceiling priority of the target mutex is lower than or equal to the ceiling priority of the locked mutexes.

Note 1 Invoking tasks are queued to the target mutex wait queue in the priority order. Among tasks with the same priority, they are queued in FIFO order.

Note 2 This service call returns "E_ILUSE" if this service call is re-issued for the mutex that has been locked by the invoking task (multiple-locking of mutex).

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	InvalidID number. - *mtxid* ≤ 0 - *mtxid* > VTMAX_MTX
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_ILUSE	-28	Illegal use of service call. - The invoking task has already locked the target mutex. - Ceiling priority violation (the base priority of the invoking task < the ceiling priority of the target mutex)
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.

ploc_mtx

Outline

Lock mutex (polling).

C format

```
ER      ploc_mtx (ID mtxid);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mtxid;`	ID number of the mutex.

Explanation

This service call locks the mutex specified by parameter *mtxid*.
If the target mutex could not be locked (another task has been locked) when this service call is issued but “E_TMOUT” is returned.
When the mutex is locked, this service call changes the current priority of the invoking task to the ceiling priority of the target mutex. However, this service call does not change the current priority when the invoking task has locked other mutexes and the ceiling priority of the target mutex is lower than or equal to the ceiling priority of the locked mutexes.

Note This service call returns “E_ILUSE” if this service call is re-issued for the mutex that has been locked by the invoking task (multiple-locking of mutex).

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *mtxid* ≤ 0 - *mtxid* > VTMAX_MTX
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the dispatching disabled state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_ILUSE	-28	Illegal use of service call. - The invoking task has already locked the target mutex. - Ceiling priority violation (the base priority of the invoking task < the ceiling priority of the target mutex)

Macro	Value	Description
E_TMOUT	-50	Polling failure.

tloc_mtx

Outline

Lock mutex (with time-out).

C format

```
ER      tloc_mtx (ID mtxid, TMO tmout);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mtxid;`	ID number of the mutex.
I	`TMO tmout;`	Specified time-out (in millisecond). TMO_FEVR: Waiting forever. TMO_POL: Polling. Value: Specified time-out.

Explanation

This service call locks the mutex specified by parameter *mtxid*.
If the target mutex could not be locked (another task has been locked) when this service call is issued, this service call queues the invoking task to the target mutex wait queue and moves it from the RUNNING state to the WAITING state with time-out (mutex wait state).
The WAITING state for a mutex is cancelled in the following cases.

WAITING State for a Mutex Cancel Operation	Return Value
The locked state of the target mutex was cancelled as a result of issuing unl_mtx.	E_OK
The locked state of the target mutex was cancelled as a result of issuing ext_tsk.	E_OK
The locked state of the target mutex was cancelled as a result of issuing ter_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 time specified by *tmout* has elapsed.	E_TMOUT

When the mutex is locked, this service call changes the current priority of the invoking task to the ceiling priority of the target mutex. However, this service call does not change the current priority when the invoking task has locked other mutexes and the ceiling priority of the target mutex is lower than or equal to the ceiling priority of the locked mutexes.

Note 1 Invoking tasks are queued to the target mutex wait queue in the priority order. Among tasks with the same priority, they are queued in FIFO order.

Note 2 This service call returns “E_ILUSE” if this service call is re-issued for the mutex that has been locked by the invoking task (multiple-locking of mutex).

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

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *tmout* < -1 - *tmout* > (0x7FFFFFFF - TIC_NUME) / TIC_DENO
E_ID	-18	Invalid ID number. - *mtxid* ≤ 0 - *mtxid* > VTMAX_MTX
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_ILUSE	-28	Illegal use of service call. - The invoking task has already locked the target mutex. - Ceiling priority violation (the base priority of the invoking task < the ceiling priority of the target mutex)
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
E_TMOUT	-50	Polling failure or specified time has elapsed.

unl_mtx

Outline

Unlock mutex.

C format

```
ER      unl_mtx (ID mtxid);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mtxid;`	ID number of the mutex.

Explanation

This service call unlocks the locked mutex specified by parameter *mtxid*.
If a task has been queued to the target mutex wait queue when this service call is issued, mutex lock processing is performed by the task (the first task in the wait queue) immediately after mutex unlock processing.
As a result, the task is unlinked from the wait queue and moves from the WAITING state (mutex wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state. And this service call changes the current priority of the task to the ceiling priority of the target mutex. However, this service call does not change the current priority when the task has locked other mutexes and the ceiling priority of the target mutex is lower than or equal to the ceiling priority of the locked mutexes.

Note 1 A locked mutex can be unlocked only by the task that locked the mutex.
If this service call is issued for a mutex that was not locked by the invoking task, "E_ILUSE" is returned.

Note 2 When a task terminates, mutexes locked by the task are unlocked.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *mtxid* ≤ 0 - *mtxid* > VTMAX_MTX
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the status "PSW.IPL > kernel interrupt mask level".
E_ILUSE	-28	Illegal use of service call. - The invoking task have not locked the target mutex.

ref_mtx

Outline

Reference mutex state.

C format

```
ER      ref_mtx (ID mtxid, T_RMTX *pk_rmtx);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mtxid;`	ID number of the mutex.
O	`T_RMTX *pk_rmtx;`	Pointer to the packet returning the mutex state.

[Mutex state packet: T_RMTX]

```
typedef struct  t_rmtx {
    ID      htskid;         /*Existence of locked mutex*/
    ID      wtskid;         /*Existence of waiting task*/
} T_RMTX;
```

Explanation

This service call stores the detailed information of the mutex specified by parameter *mtxid* (existence of locked mutexes, waiting tasks, etc.) into the area specified by parameter *pk_rmtx*.

- *htskid*
  Stores whether a task that is locking a mutex exists.
  TSK_NONE: No applicable task
  Value: ID number of the task locking the mutex

- *wtskid*
  Stores whether a task is queued to the mutex wait queue.
  TSK_NONE: No applicable task
  Value: ID number of the task at the head of the wait queue

Return value

Macro	Value	Description
E_OK	0	Normal completion.

Macro	Value	Description
E_ID	-18	Invalid ID number. - *mtxid* ≤ 0 - *mtxid* > VTMAX_MTX
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.

18.2.8 Extended synchronization and communication functions (message buffers)

The following shows the service calls provided by the RI600V4 as the extended synchronization and communication functions (message buffers).

Table 18-10 Extended Synchronization and Communication Functions (Message Buffers)

Service Call	Function	Useful Range
snd_mbf	Send to message buffer (waiting forever)	Task
psnd_mbf	Send to message buffer (polling)	Task
ipsnd_mbf	Send to message buffer (polling)	Non-task
tsnd_mbf	Send to message buffer (with time-out)	Task
rcv_mbf	Receive from message buffer (waiting forever)	Task
prcv_mbf	Receive from message buffer (polling)	Task
trcv_mbf	Receive from message buffer (with time-out)	Task
ref_mbf	Reference message buffer state	Task
iref_mbf	Reference message buffer state	Non-task

snd_mbf

Outline

Send to message buffer (waiting forever).

C format

```
ER      tsnd_mbf (ID mbfid, VP msg, UINT msgsz);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mbfid;`	ID number of the message buffer.
I	`VP msg;`	Pointer to the message to be sent.
I	`UINT msgsz;`	Message size to be sent (in bytes).

Explanation

This service call processes as follows according to the situation of the message buffer specified by the parameter *mbfid*.

- There is a task in the reception wait queue.
  This service call transfers the message specified by parameter *msg* to the task in the top of the reception wait queue. As a result, the task is unlinked from the reception wait queue and moves from the WAITING state (message reception wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

- There is no task neither in the reception wait queue and transmission wait queue and there is available space in the message buffer.
  This service call stores the message specified by parameter *msg* to the message buffer. As a result, the size of available space in the target message buffer decreases by the amount calculated by the following expression.
  The amount of decrease = up4(*msgsz*) + VTSZ_MBFTBL

- There is no task neither in the reception wait queue and transmission wait queue and there is no available space in the message buffer, or there is a task in the transmission wait queue.
  This service call queues the invoking task to the transmission wait queue of the target message buffer and moves it from the RUNNING state to the WAITING state (message transmission wait state).
  The sending WAITING state for a message buffer is cancelled in the following cases.

Sending WAITING State for a Message Buffer Cancel Operation	Return Value
Available space was secured in the message buffer area as a result of issuing rcv_mbf.	E_OK
Available space was secured in the message buffer area as a result of issuing prcv_mbf.	E_OK
Available space was secured in the message buffer area as a result of issuing trcv_mbf.	E_OK

Sending WAITING State for a Message Buffer Cancel Operation	Return Value
The task at the top of the transmission wait queue was forcedly released from waiting by following either. - Forced release from waiting (accept rel_wai while waiting). - Forced release from waiting (accept irel_wai while waiting). - Forced release from waiting (accept ter_tsk while waiting). - The time specified by *tmout* for tsnd_mbf 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 message buffer is reset as a result of issuing vrst_mbf.	EV_RST

Note 1 Message is written to the message buffer area in the order of the message transmission request.

Note 2 Invoking tasks are queued to the transmission wait queue of the target message buffer in the FIFO order.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *msgsz* == 0 - *msgsz* > Maximum message size (max_msgsz)
E_ID	-18	Invalid ID number. - *mbfid* ≤ 0 - *mbfid* > VTMAX_MBF
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
EV_RST	-127	Released from WAITING state by the object reset (vrst_mbf)

psnd_mbf
ipsnd_mbf

Outline

Send to message buffer (polling).

C format

```
ER      psnd_mbf (ID mbfid, VP msg, UINT msgsz);
ER      ipsnd_mbf (ID mbfid, VP msg, UINT msgsz);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mbfid;`	ID number of the message buffer.
I	`VP msg;`	Pointer to the message to be sent.
I	`UINT msgsz;`	Message size to be sent (in bytes).

Explanation

These service calls process as follows according to the situation of the message buffer specified by the parameter *mbfid*.

- There is a task in the reception wait queue.
  This service call transfers the message specified by parameter *msg* to the task in the top of the reception wait queue. As a result, the task is unlinked from the reception wait queue and moves from the WAITING state (message reception wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

- There is no task neither in the reception wait queue and transmission wait queue and there is available space in the message buffer.
  This service call stores the message specified by parameter *msg* to the message buffer. As a result, the size of available space in the target message buffer decreases by the amount calculated by the following expression.

  The amount of decrease = up4(*msgsz*) + VTSZ_MBFTBL

- There is no task neither in the reception wait queue and transmission wait queue and there is no available space in the message buffer, or there is a task in the transmission wait queue.
  These service calls return “E_TMOUT”.

Note Message is written to the message buffer area in the order of the message transmission request.

Return value

Macro	Value	Description
E_OK	0	Normal completion.

Macro	Value	Description
E_PAR	-17	Parameter error. - *msgsz* == 0 - *msgsz* > Maximum message size (max_msgsz)
E_ID	-18	Invalid ID number. - *mbfid* ≤ 0 - *mbfid* > VTMAX_MBF
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - The ipsnd_mbf was issued from task. - The psnd_mbf was issued from non-task. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_TMOUT	-50	Polling failure.

tsnd_mbf

Outline

Send to message buffer (with time-out).

C format

```
ER      tsnd_mbf (ID mbfid, VP msg, UINT msgsz, TMO tmout);
```

Parameter(s)

I/O	Parameter	Description
I	ID *mbfid;*	ID number of the message buffer.
I	VP *msg;*	Pointer to the message to be sent.
I	UINT *msgsz;*	Message size to be sent (in bytes).
I	TMO *tmout;*	Specified time-out (in millisecond). TMO_FEVR: Waiting forever. TMO_POL: Polling. Value: Specified time-out.

Explanation

This service call processes as follows according to the situation of the message buffer specified by the parameter *mbfid*.

- There is a task in the reception wait queue.
  This service call transfers the message specified by parameter *msg* to the task in the top of the reception wait queue. As a result, the task is unlinked from the reception wait queue and moves from the WAITING state (message reception wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

- There is no task neither in the reception wait queue and transmission wait queue and there is available space in the message buffer.
  This service call stores the message specified by parameter *msg* to the message buffer. As a result, the size of available space in the target message buffer decreases by the amount calculated by the following expression.

  The amount of decrease = up4(*msgsz*) + VTSZ_MBFTBL

- There is no task neither in the reception wait queue and transmission wait queue and there is no available space in the message buffer, or there is a task in the transmission wait queue.
  This service call queues the invoking task to the transmission wait queue of the target message buffer and moves it from the RUNNING state to the WAITING state with time (message transmission wait state).
  The sending WAITING state for a message buffer is cancelled in the following cases.

Sending WAITING State for a Message Buffer Cancel Operation	Return Value
Available space was secured in the message buffer area as a result of issuing rcv_mbf.	E_OK
Available space was secured in the message buffer area as a result of issuing prcv_mbf.	E_OK
Available space was secured in the message buffer area as a result of issuing trcv_mbf.	E_OK

Sending WAITING State for a Message Buffer Cancel Operation	Return Value
The task at the top of the transmission wait queue was forcedly released from waiting by following either. - Forced release from waiting (accept rel_wai while waiting). - Forced release from waiting (accept irel_wai while waiting). - Forced release from waiting (accept ter_tsk while waiting). - The time specified by *tmout* for tsnd_mbf 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 message buffer is reset as a result of issuing vrst_mbf.	EV_RST
The time specified by *tmout* has elapsed.	E_TMOUT

Note 1 Message is written to the message buffer area in the order of the message transmission request.

Note 2 Invoking tasks are queued to the transmission wait queue of the target message buffer in the FIFO order.

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

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *msgsz* == 0 - *msgsz* > Maximum message size (max_msgsz) - *tmout* < -1 - *tmout* > (0x7FFFFFFF - TIC_NUME) / TIC_DENO
E_ID	-18	Invalid ID number. - *mbfid* ≤ 0 - *mbfid* > VTMAX_MBF
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
E_TMOUT	-50	Polling failure or specified time has elapsed.
EV_RST	-127	Released from WAITING state by the object reset (vrst_mbf)

rcv_mbf

Outline

Receive from message buffer (waiting forever).

C format

```
ER_UINT   rcv_mbf (ID mbfid, VP msg);
```

Parameter(s)

I/O	Parameter	Description
I	`ID        mbfid;`	ID number of the message buffer.
O	`VP        msg;`	Pointer to store the message.

Explanation

This service call processes as follows according to the situation of the message buffer specified by the parameter *mbfid*.

- There is a message in the message buffer.
  This service call takes out the oldest message from the message buffer and stores the message to the area specified by *msg* and return the size of the message. As a result, the size of available space in the target message buffer increases by the amount calculated by the following expression.
  The amount of increase = up4(Return value) + VTSZ_MBFTBL

  In addition, this service call repeats the following processing until task in the transmission wait queue is lost or it becomes impossible to store the message in the message buffer.
  - When there is a task in the transmission wait queue and there is available space in the message buffer for the message specified by the task in the top of the transmission wait queue, the task is unlinked from the transmission wait queue and moves from the WAITING state (message transmission wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state. As a result, the size of available space in the target message buffer decreases by the amount calculated by the following expression.
    The amount of decrease =up4(The message size sent by the task) + VTSZ_MBFTBL
- There is no message in the message buffer and there is a task in the transmission wait queue.
  This service call stores the message specified by the task in the top of the transmission wait queue to the area pointed by the parameter *msg*. As a result, the task is unlinked from the transmission wait queue and moves from the WAITING state (message transmission wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  Note, this situation is caused only when the size of the message buffer is 0.
- There is no message in the message buffer and there is no task in the transmission wait queue.
  This service call queues the invoking task to the reception wait queue of the target message buffer and moves it from the RUNNING state to the WAITING state (message reception wait state).
  The receiving WAITING state for a message buffer is cancelled in the following cases.

Receiving WAITING State for a Message Buffer Cancel Operation	Return Value
Message was sent to the message buffer area as a result of issuing snd_mbf.	E_OK

Receiving WAITING State for a Message Buffer Cancel Operation	Return Value
Message was sent to the message buffer area as a result of issuing psnd_mbf.	E_OK
Message was sent to the message buffer area as a result of issuing ipsnd_mbf.	E_OK
Message was sent to the message buffer area as a result of issuing tsnd_mbf.	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 The Maximum message size (max_msgsz) is defined during configuration. The size of the area pointed by msg must be larger than or equal to the maximum message size.

Note 2 Invoking tasks are queued to the reception wait queue of the target message buffer in the order of the message reception request.

Return value

Macro	Value	Description
E_ID	-18	Invalid ID number. - *mbfid* ≤ 0 - *mbfid* > VTMAX_MBF
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
-	Positive value	Normal completion (the size of the received message).

prcv_mbf

Outline

Receive from message buffer (polling).

C format

```
ER_UINT  prcv_mbf (ID mbfid, VP msg);
```

Parameter(s)

I/O	Parameter	Description
I	`ID` *`mbfid;`*	ID number of the message buffer.
O	`VP` *`msg;`*	Pointer to store the message.

Explanation

- There is a message in the message buffer.
  This service call takes out the oldest message from the message buffer and stores the message to the area specified by *msg* and return the size of the message. As a result, the size of available space in the target message buffer increases by the amount calculated by the following expression.

  The amount of increase = up4(Return value) + VTSZ_MBFTBL

  In addition, this service call repeats the following processing until task in the transmission wait queue is lost or it becomes impossible to store the message in the message buffer.

  - When there is a task in the transmission wait queue and there is available space in the message buffer for the message specified by the task in the top of the transmission wait queue, the task is unlinked from the transmission wait queue and moves from the WAITING state (message transmission wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state. As a result, the size of available space in the target message buffer decreases by the amount calculated by the following expression.

    The amount of decrease =up4(The message size sent by the task) + VTSZ_MBFTBL

- There is no message in the message buffer and there is a task in the transmission wait queue.
  This service call stores the message specified by the task in the top of the transmission wait queue to the area pointed by the parameter *msg*. As a result, the task is unlinked from the transmission wait queue and moves from the WAITING state (message transmission wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  Note, this situation is caused only when the size of the message buffer is 0.

- There is no message in the message buffer and there is no task in the transmission wait queue.
  This service call returns “E_TMOUT”.

Note The Maximum message size (max_msgsz) is defined during configuration. The size of the area pointed by msg must be larger than or equal to the maximum message size.

Return value

Macro	Value	Description
E_ID	-18	Invalid ID number. - *mbfid* ≤ 0 - *mbfid* > VTMAX_MBF
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_TMOUT	-50	Polling failure.
-	Positive value	Normal completion (the size of the received message).

trcv_mbf

Outline

Receive from message buffer (with time-out).

C format

```
ER_UINT  trcv_mbf (ID mbfid, VP msg, TMO tmout);
```

Parameter(s)

I/O	Parameter	Description
I	ID *mbfid;*	ID number of the message buffer.
O	VP *msg;*	Pointer to store the message.
I	TMO *tmout;*	Specified time-out (in millisecond). TMO_FEVR: Waiting forever. TMO_POL: Polling. Value: Specified time-out.

Explanation

This service call processes as follows according to the situation of the message buffer specified by the parameter *mbfid*.

- There is a message in the message buffer.
  This service call takes out the oldest message from the message buffer and stores the message to the area specified by *msg* and return the size of the message. As a result, the size of available space in the target message buffer increases by the amount calculated by the following expression.

  The amount of increase = up4(Return value) + VTSZ_MBFTBL

  In addition, this service call repeats the following processing until task in the transmission wait queue is lost or it becomes impossible to store the message in the message buffer.

  - When there is a task in the transmission wait queue and there is available space in the message buffer for the message specified by the task in the top of the transmission wait queue, the task is unlinked from the transmission wait queue and moves from the WAITING state (message transmission wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state. As a result, the size of available space in the target message buffer decreases by the amount calculated by the following expression.

    The amount of decrease =up4(The message size sent by the task) + VTSZ_MBFTBL

- There is no message in the message buffer and there is a task in the transmission wait queue.
  This service call stores the message specified by the task in the top of the transmission wait queue to the area pointed by the parameter *msg*. As a result, the task is unlinked from the transmission wait queue and moves from the WAITING state (message transmission wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  Note, this situation is caused only when the size of the message buffer is 0.

- There is no message in the message buffer and there is no task in the transmission wait queue.
  This service call queues the invoking task to the reception wait queue of the target message buffer and moves it from the RUNNING state to the WAITING state with time (message reception wait state).
  The receiving WAITING state for a message buffer is cancelled in the following cases.

Receiving WAITING State for a Message Buffer Cancel Operation	Return Value
Message was sent to the message buffer area as a result of issuing snd_mbf.	E_OK
Message was sent to the message buffer area as a result of issuing psnd_mbf.	E_OK
Message was sent to the message buffer area as a result of issuing ipsnd_mbf.	E_OK
Message was sent to the message buffer area as a result of issuing tsnd_mbf.	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 time specified by *tmout* has elapsed.	E_TMOUT

Note 1 The Maximum message size (max_msgsz) is defined during configuration. The size of the area pointed by msg must be larger than or equal to the maximum message size.

Note 2 Invoking tasks are queued to the reception wait queue of the target message buffer in the order of the message reception request.

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

Return value

Macro	Value	Description
E_PAR	-17	Parameter error. - *tmout* < -1 - *tmout* > (0x7FFFFFFF - TIC_NUME) / TIC_DENO
E_ID	-18	Invalid ID number. - *mbfid* ≤ 0 - *mbfid* > VTMAX_MBF
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
E_TMOUT	-50	Polling failure or specified time has elapsed.
-	Positive value	Normal completion (the size of the received message).

ref_mbf
iref_mbf

Outline

Reference message buffer state.

C format

```
ER      ref_mbf (ID mbfid, T_RMBF *pk_rmbf);
ER      iref_mbf (ID mbfid, T_RMBF *pk_rmbf);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mbfid;`	ID number of the message.
O	`T_RMBF *pk_rmbf;`	Pointer to the packet returning the message buffer state.

[Message buffer state packet: T_RMBF]

```
typedef struct  t_rmbf {
    ID      stskid;         /*Existence of tasks waiting for message transmission*/
    ID      rtskid;         /*Existence of tasks waiting for message reception*/
    UINT    smsgcnt;        /*Number of message elements in message buffer*/
    SIZE    fmbfsz;         /*Available buffer size*/
} T_RMBF;
```

Explanation

These service calls store the detailed information of the message buffer (existence of waiting tasks, number of data elements in the message buffer, etc.) specified by parameter *mbfid* into the area specified by parameter *pk_rmbf*.

- *stskid*
  Stores whether a task is queued to the transmission wait queue of the message buffer.

  TSK_NONE: No applicable task
  Value: ID number of the task at the head of the transmission wait queue

- *rtskid*
  Stores whether a task is queued to the reception wait queue of the message buffer.

  TSK_NONE: No applicable task
  Value: ID number of the task at the head of the reception wait queue

- *smsgcnt*
  Stores the number of message elements in message buffer.

- *fmbfsz*
  Stores available size of the message buffer (in bytes).

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *mbfid* ≤ 0 - *mbfid* > VTMAX_MBF
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”. Note When the iref_mbf is issued from task or the ref_mbf is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.

18.2.9 Memory pool management functions (fixed-sized memory pools)

The following shows the service calls provided by the RI600V4 as the memory pool management functions (fixed-sized memory pools).

Table 18-11 Memory Pool Management Functions (Fixed-Sized Memory Pools)

Service Call	Function	Useful Range
get_mpf	Acquire fixed-sized memory block (waiting forever)	Task
pget_mpf	Acquire fixed-sized memory block (polling)	Task
ipget_mpf	Acquire fixed-sized memory block (polling)	Non-task
tget_mpf	Acquire fixed-sized memory block (with time-out)	Task
rel_mpf	Release fixed-sized memory block	Task
irel_mpf	Release fixed-sized memory block	Non-task
ref_mpf	Reference fixed-sized memory pool state	Task
iref_mpf	Reference fixed-sized memory pool state	Non-task

get_mpf

Outline

Acquire fixed-sized memory block (waiting forever).

C format

```
ER      get_mpf (ID mpfid, VP *p_blk);
```

Parameter(s)

I/O	Parameter	Description
I	ID *mpfid;*	ID number of the fixed-sized memory pool.
O	VP **p_blk;*	Start address of the acquired memory block.

Explanation

This service call acquires the fixed-sized 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 no fixed-size memory blocks could be acquired from the target fixed-size memory pool (no available fixed-size memory blocks exist) when this service call is issued, this service call does not acquire the fixed-size memory block but queues the invoking task to the target fixed-size memory pool wait queue and moves it from the RUNNING state to the WAITING state (fixed-size memory block acquisition wait state).
The WAITING state for a fixed-sized memory block is cancelled in the following cases.

WAITING State for a Fixed-sized Memory Block Cancel Operation	Return Value
A fixed-sized memory block was returned to the target fixed-sized memory pool as a result of issuing rel_mpf.	E_OK
A fixed-sized memory block was returned to the target fixed-sized memory pool as a result of issuing irel_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 fixed-sized memory pool is reset as a result of issuing vrst_mpf.	EV_RST

Note 1 Invoking tasks are queued to the target fixed-size memory pool wait queue in the order defined during configuration (FIFO order or current priority order).

Note 2 The contents of the block are undefined.

Note 3 The boundary alignment for the memory blocks acquired is 1. If memory blocks need to be acquired with a larger boundary alignment than that, observe the following:

- Set The size of the fixed-sized memory block (siz_block) in Fixed-sized Memory Pool Information (memorypool[]) to multiple of the desired boundary alignment.
- Specify unique section name to the Section name assigned to the memory pool area (section) in Fixed-sized Memory Pool Information (memorypool[]) and locate the section to the address of the desired boundary alignment when linking.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *mpfid* ≤ 0 - *mpfid* > VTMAX_MPF
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
EV_RST	-127	Released from WAITING state by the object reset (vrst_mpf)

pget_mpf
ipget_mpf

Outline

Acquire fixed-sized memory block (polling).

C format

```
ER      pget_mpf (ID mpfid, VP *p_blk);
ER      ipget_mpf (ID mpfid, VP *p_blk);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mpfid;`	ID number of the fixed-sized memory pool.
O	`VP *p_blk;`	Start address of the acquired memory block.

Explanation

This service call acquires the fixed-sized 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 fixed-sized memory block could not be acquired from the target fixed-sized memory pool (no available fixed-sized memory blocks exist) when this service call is issued, fixed-sized memory block acquisition processing is not performed but “E_TMOUT” is returned.

Note 1 The contents of the block are undefined.

Note 2 The boundary alignment for the memory blocks acquired is 1. If memory blocks need to be acquired with a larger boundary alignment than that, observe the following:

- Set The size of the fixed-sized memory block (siz_block) in Fixed-sized Memory Pool Information (memorypool[]) to multiple of the desired boundary alignment.
- Specify unique section name to the Section name assigned to the memory pool area (section) in Fixed-sized Memory Pool Information (memorypool[]) and locate the section to the address of the desired boundary alignment when linking.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *mpfid* ≤ 0 - *mpfid* > VTMAX_MPF

Macro	Value	Description
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status "PSW.IPL > kernel interrupt mask level". Note When the ipget_mpf is issued from task or the pget_mpf is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.
E_TMOUT	-50	Polling failure.

tget_mpf

Outline

Acquire fixed-sized memory block (with time-out).

C format

```
ER      tget_mpf (ID mpfid, VP *p_blk, TMO tmout);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mpfid;`	ID number of the fixed-sized memory pool.
O	`VP *p_blk;`	Start address of the acquired memory block.
I	`TMO tmout;`	Specified time-out (in millisecond). TMO_FEVR: Waiting forever. TMO_POL: Polling. Value: Specified time-out.

Explanation

This service call acquires the fixed-sized 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 no fixed-size memory blocks could be acquired from the target fixed-size memory pool (no available fixed-size memory blocks exist) when this service call is issued, this service call does not acquire the fixed-size memory block but queues the invoking task to the target fixed-size memory pool wait queue and moves it from the RUNNING state to the WAITING state with time-out (fixed-size memory block acquisition wait state).
The WAITING state for a fixed-sized memory block is cancelled in the following cases.

WAITING State for a Fixed-sized Memory Block Cancel Operation	Return Value
A fixed-sized memory block was returned to the target fixed-sized memory pool as a result of issuing rel_mpf.	E_OK
A fixed-sized memory block was returned to the target fixed-sized memory pool as a result of issuing irel_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 fixed-sized memory pool is reset as a result of issuing vrst_mpf.	EV_RST
The time specified by *tmout* has elapsed.	E_TMOUT

Note 1 Invoking tasks are queued to the target fixed-size memory pool wait queue in the order defined during configuration (FIFO order or current priority order).

Note 2 The contents of the block are undefined.

Note 3 The boundary alignment for the memory blocks acquired is 1. If memory blocks need to be acquired with a larger boundary alignment than that, observe the following:

- Set The size of the fixed-sized memory block (siz_block) in Fixed-sized Memory Pool Information (memorypool[]) to multiple of the desired boundary alignment.
- Specify unique section name to the Section name assigned to the memory pool area (section) in Fixed-sized Memory Pool Information (memorypool[]) and locate the section to the address of the desired boundary alignment when linking.

Note 4 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_PAR	-17	Parameter error. - *tmout* < -1 - *tmout* > (0x7FFFFFFF - TIC_NUME) / TIC_DENO
E_ID	-18	Invalid ID number. - *mpfid* ≤ 0 - *mpfid* > VTMAX_MPF
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
E_TMOUT	-50	Polling failure or specified time has elapsed.
EV_RST	-127	Released from WAITING state by the object reset (vrst_mpf)

rel_mpf
irel_mpf

Outline

Release fixed-sized memory block.

C format

```
ER      rel_mpf (ID mpfid, VP blk);
ER      irel_mpf (ID mpfid, VP blk);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mpfid;`	ID number of the fixed-sized memory pool.
I	`VP blk;`	Start address of the memory block to be released.

Explanation

This service call returns the fixed-sized 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, fixed-sized memory block return processing is not performed but fixed-sized 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.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *blk* is illegal.
E_ID	-18	Invalid ID number. - *mpfid* ≤ 0 - *mpfid* > VTMAX_MPF

Macro	Value	Description
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - The irel_mpf was issued from task. - The rel_mpf was issued from non-task. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.

ref_mpf
iref_mpf

Outline

Reference fixed-sized memory pool state.

C format

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

Parameter(s)

I/O	Parameter	Description
I	`ID mpfid;`	ID number of the fixed-sized memory pool.
O	`T_RMPF *pk_rmpf;`	Pointer to the packet returning the fixed-sized memory pool state.

[Fixed-sized memory pool state packet: T_RMPF]

```
typedef struct  t_rmpf {
    ID      wtskid;         /*Existence of waiting task*/
    UINT    fblkcnt;        /*Number of free memory blocks*/
} T_RMPF;
```

Explanation

Stores fixed-sized memory pool state packet (ID number of the task at the head of the wait queue, number of free memory blocks, etc.) of the fixed-sized memory pool specified by parameter *mpfid* in the area specified by parameter *pk_rmpf*.

- *wtskid*
  Stores whether a task is queued to the fixed-size memory pool.

TSK_NONE:	No applicable task
Value:	ID number of the task at the head of the wait queue

- *fblkcnt*
  Stores the number of free memory blocks.

Return value

Macro	Value	Description
E_OK	0	Normal completion.

Macro	Value	Description
E_ID	-18	Invalid ID number. - *mpfid* ≤ 0 - *mpfid* > VTMAX_MPF
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”. Note When the iref_mpf is issued from task or the ref_mpf is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.

18.2.10 Memory pool management functions (variable-sized memory pools)

The following shows the service calls provided by the RI600V4 as the memory pool management functions (variable-sized memory pools).

Table 18-12 Memory Pool Management Functions (Variable-Sized Memory Pools)

Service Call	Function	Useful Range
get_mpl	Acquire variable-sized memory block (waiting forever)	Task
pget_mpl	Acquire variable-sized memory block (polling)	Task
ipget_mpl	Acquire variable-sized memory block (polling)	Non-task
tget_mpl	Acquire variable-sized memory block (with time-out)	Task
rel_mpl	Release variable-sized memory block	Task
ref_mpl	Reference variable-sized memory pool state	Task
iref_mpl	Reference variable-sized memory pool state	Non-task

get_mpl

Outline

Acquire variable-sized memory block (waiting forever).

C format

```
ER      get_mpl (ID mplid, UINT blksz, VP *p_blk);
```

Parameter(s)

I/O	Parameter	Description
I	ID *mplid;*	ID number of the variable-sized memory pool.
I	UINT *blksz;*	Memory block size to be acquired (in bytes).
O	VP **p_blk;*	Start address of the acquired memory block.

Explanation

This service call acquires a variable-size memory block of the size specified by parameter blksz from the variable-size memory pool specified by parameter *mplid*, and stores its start address into the area specified by parameter *p_blk*.
If no variable-size memory blocks could be acquired from the target variable-size memory pool (no successive areas equivalent to the requested size were available) when this service call is issued, this service call does not acquire variable-size memory blocks but queues the invoking task to the target variable-size memory pool wait queue and moves it from the RUNNING state to the WAITING state (variable-size memory block acquisition wait state).
The WAITING state for a variable-sized memory block is cancelled in the following cases.

WAITING State for a Variable-sized Memory Block Cancel Operation	Return Value
The variable-size memory block that satisfies the requested size was returned to the target variable-size memory pool as a result of issuing rel_mpl.	E_OK
The task at the top of the transmission wait queue was forcedly released from waiting by following either. - Forced release from waiting (accept rel_wai while waiting). - Forced release from waiting (accept irel_wai while waiting). - Forced release from waiting (accept ter_tsk while waiting). - The time specified by *tmout* for tget_mpl 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 variable-sized memory pool is reset as a result of issuing vrst_mpl.	EV_RST

Note 1 For the size of the memory block, refer to "7.3.2 Size of Variable-sized memory block.".

Note 2 Invoking tasks are queued to the target variable-size memory pool wait queue in the FIFO order.

Note 3 The contents of the block are undefined.

Note 4 The alignment number of memory blocks is 1. To enlarge the alignment number to 4, specify unique section to Section name assigned to the memory pool area (mpl_section) in Variable-sized Memory Pool Information (variable_memorypool[]) and locate the section to 4-bytes boundary address when linking.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *blksz* == 0 - *blksz* exceeds the maximum size that can be acquired.
E_ID	-18	Invalid ID number. - *mplid* $\leq$ 0 - *mplid* > VTMAX_MPL
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
EV_RST	-127	Released from WAITING state by the object reset (vrst_mpl)

pget_mpl
ipget_mpl

Outline

Acquire variable-sized memory block (polling).

C format

```
ER      pget_mpl (ID mplid, UINT blksz, VP *p_blk);
ER      ipget_mpl (ID mplid, UINT blksz, VP *p_blk);
```

Parameter(s)

I/O	Parameter	Description
I	`ID` *mplid;*	ID number of the variable-sized memory pool.
I	`UINT` *blksz;*	Memory block size to be acquired (in bytes).
O	`VP` **p_blk;*	Start address of the acquired memory block.

Explanation

This service call acquires a variable-size memory block of the size specified by parameter *blksz* from the variable-size memory pool specified by parameter *mplid*, and stores its start address into the area specified by parameter *p_blk*.
If no variable-size memory blocks could be acquired from the target variable-size memory pool (no successive areas equivalent to the requested size were available) when this service call is issued, this service call does not acquire variable-size memory block but returns “E_TMOUT”.

Note 1 For the size of the memory block, refer to “7.3.2 Size of Variable-sized memory block.”.

Note 2 The contents of the block are undefined.

Note 3 The alignment number of memory blocks is 1. To enlarge the alignment number to 4, specify unique section to Section name assigned to the memory pool area (mpl_section) in Variable-sized Memory Pool Information (variable_memorypool[]) and locate the section to 4-bytes boundary address when linking.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *blksz* == 0 - *blksz* exceeds the maximum size that can be acquired.
E_ID	-18	Invalid ID number. - *mplid* ≤ 0 - *mplid* > VTMAX_MPL

Macro	Value	Description
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status "PSW.IPL > kernel interrupt mask level". Note When the ipget_mpl is issued from task or the pget_mpl is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.
E_TMOUT	-50	Polling failure.

tget_mpl

Outline

Acquire variable-sized memory block (with time-out).

C format

```
ER      tget_mpl (ID mplid, UINT blksz, VP *p_blk, TMO tmout);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mplid;`	ID number of the variable-sized memory pool.
I	`UINT blksz;`	Memory block size to be acquired (in bytes).
O	`VP *p_blk;`	Start address of the acquired memory block.
I	`TMO tmout;`	Specified time-out (in millisecond). TMO_FEVR: Waiting forever. TMO_POL: Polling. Value: Specified time-out.

Explanation

This service call acquires a variable-size memory block of the size specified by parameter *blksz* from the variable-size memory pool specified by parameter *mplid*, and stores its start address into the area specified by parameter *p_blk*.
If no variable-size memory blocks could be acquired from the target variable-size memory pool (no successive areas equivalent to the requested size were available) when this service call is issued, this service call does not acquire variable-size memory blocks but queues the invoking task to the target variable-size memory pool wait queue and moves it from the RUNNING state to the WAITING state with time-out (variable-size memory block acquisition wait state).
The WAITING state for a variable-sized memory block is cancelled in the following cases.

WAITING State for a Variable-sized Memory Block Cancel Operation	Return Value
The variable-size memory block that satisfies the requested size was returned to the target variable-size memory pool as a result of issuing rel_mpl.	E_OK
The task at the top of the transmission wait queue was forcedly released from waiting by following either. - Forced release from waiting (accept rel_wai while waiting). - Forced release from waiting (accept irel_wai while waiting). - Forced release from waiting (accept ter_tsk while waiting). - The time specified by *tmout* for tget_mpl 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 variable-sized memory pool is reset as a result of issuing vrst_mpl.	EV_RST
The time specified by *tmout* has elapsed.	E_TMOUT

Note 1 For the size of the memory block, refer to “7.3.2 Size of Variable-sized memory block.”.

Note 2 Invoking tasks are queued to the target variable-size memory pool wait queue in the FIFO order.

Note 3 The contents of the block are undefined.

Note 4 The alignment number of memory blocks is 1. To enlarge the alignment number to 4, specify unique section to Section name assigned to the memory pool area (mpl_section) in Variable-sized Memory Pool Information (variable_memorypool[]) and locate the section to 4-bytes boundary address when linking.

Note 5 TMO_FEVR is specified for wait time *tmout*, processing equivalent to get_mpl will be executed. When TMO_POL is specified, processing equivalent to pget_mpl will be executed.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *blksz* == 0 - *blksz* exceeds the maximum size that can be acquired. - *tmout* < -1 - *tmout* > (0x7FFFFFFF - TIC_NUME) / TIC_DENO
E_ID	-18	Invalid ID number. - *mplid* ≤ 0 - *mplid* > VTMAX_MPL
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
E_TMOUT	-50	Polling failure or specified time has elapsed.
EV_RST	-127	Released from WAITING state by the object reset (vrst_mpl)

rel_mpl

Outline

Release variable-sized memory block.

C format

```
ER       rel_mpl (ID mplid, VP blk);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mplid;`	ID number of the variable-sized memory pool.
I	`VP blk;`	Start address of memory block to be released.

Explanation

This service call returns the variable-sized memory block specified by parameter *blk* to the variable-sized memory pool specified by parameter *mplid*.
After returning the variable-size memory blocks, these service calls check the tasks queued to the target variable-size memory pool wait queue from the top, and assigns the memory if the size of memory requested by the wait queue is available. This operation continues until no tasks queued to the wait queue remain or no memory space is available. As a result, the task that acquired the memory is unlinked from the queue and moved from the WAITING state (variable-size memory block acquisition wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

Note The RI600V4 do only simple error detection for *blk*. If *blk* is illegal and the error is not detected, the operation is not guaranteed after that.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *blk* is illegal.
E_ID	-18	Invalid ID number. - *mplid* ≤ 0 - *mplid* > VTMAX_MPL

Macro	Value	Description
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued from non-task. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.

ref_mpl
iref_mpl

Outline

Reference variable-sized memory pool state.

C format

```
ER      ref_mpl (ID mplid, T_RMPL *pk_rmpl);
ER      iref_mpl (ID mplid, T_RMPL *pk_rmpl);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mplid;`	ID number of the variable-sized memory pool.
O	`T_RMPL *pk_rmpl;`	Pointer to the packet returning the variable-sized memory pool state.

[Variable-sized memory pool state packet: T_RMPL]

```
typedef struct  t_rmpl {
    ID      wtskid;         /*Existence of waiting task*/
    SIZE    fmplsz;         /*Total size of free memory blocks*/
    UINT    fblksz;         /*Maximum memory block size available*/
} T_RMPL;
```

Explanation

These service calls store the detailed information (ID number of the task at the head of the wait queue, total size of free memory blocks, etc.) of the variable-size memory pool specified by parameter *mplid* into the area specified by parameter *pk_rmpl*.

- *wtskid*
  Stores whether a task is queued to the variable-size memory pool wait queue.

  TSK_NONE: No applicable task
  Value: ID number of the task at the head of the wait queue

- *fmplsz*
  Stores the total size of free memory blocks (in bytes).

- *fblksz*
  Stores the maximum memory block size available (in bytes).

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *mplid* ≤ 0 - *mplid* > VTMAX_MPL
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”. Note When the iref_mpl is issued from task or the ref_mpl is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.

18.2.11 Time management functions

The following shows the service calls provided by the RI600V4 as the time management functions.

Table 18-13 Time Management Functions

Service Call	Function	Useful Range
set_tim	Set system time	Task
iset_tim	Set system time	Non-task
get_tim	Reference system time	Task
iget_tim	Reference system time	Non-task
sta_cyc	Start cyclic handler operation	Task
ista_cyc	Start cyclic handler operation	Non-task
stp_cyc	Stop cyclic handler operation	Task
istp_cyc	Stop cyclic handler operation	Non-task
ref_cyc	Reference cyclic handler state	Task
iref_cyc	Reference cyclic handler state	Non-task
sta_alm	Start alarm handler operation	Task
ista_alm	Start alarm handler operation	Non-task
stp_alm	Stop alarm handler operation	Task
istp_alm	Stop alarm handler operation	Non-task
ref_alm	Reference alarm handler state	Task
iref_alm	Reference alarm handler state	Non-task

set_tim
iset_tim

Outline

Set system time.

C format

```
ER      set_tim (SYSTIM *p_systim);
ER      iset_tim (SYSTIM *p_systim);
```

Parameter(s)

I/O	Parameter	Description
I	`SYSTIM *p_systim;`	Time to set as system time.

[System time packet: SYSTIM]

```
typedef struct  systim {
    UH      utime;          /*System time (higher 16 bits)*/
    UW      ltime;          /*System time (lower 32 bits)*/
} SYSTIM;
```

Explanation

These service calls change the RI600V4 system time (unit: msec) to the time specified by parameter *p_systim*.

Note Even if the system time is changed, the actual time at which the time management requests made before that (e.g., task time-outs, task delay by dly_tsk, cyclic handlers, and alarm handlers) are generated will not change.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status "PSW.IPL > kernel interrupt mask level". Note When the iset_tim is issued from task or the set_tim is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.

get_tim
iget_tim

Outline

Reference system time.

C format

```
ER      get_tim (SYSTIM *p_systim);
ER      iget_tim (SYSTIM *p_systim);
```

Parameter(s)

I/O	Parameter	Description
O	`SYSTIM *p_systim;`	Current system time.

[System time packet: SYSTIM]

```
typedef struct  systim {
    UH      utime;          /*System time (higher 16 bits)*/
    UW      ltime;          /*System time (lower 32 bits)*/
} SYSTIM;
```

Explanation

These service calls store the RI600V4 system time (unit: msec) into the area specified by parameter *p_systim*.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”. Note When the iget_tim is issued from task or the get_tim is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.

sta_cyc
ista_cyc

Outline

Start cyclic handler operation.

C format

```
ER      sta_cyc (ID cycid);
ER      ista_cyc (ID cycid);
```

Parameter(s)

I/O	Parameter	Description
I	`ID cycid;`	ID number of the cyclic handler.

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 RI600V4.
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 (phsatr) is specified for the target cyclic handler during configuration. For details, refer to "8.6.4 Start cyclic handler operation".

- When the TA_PHS attribute is specified
  The target cyclic handler activation timing is set up according to Activation phase (phs_counter) and Activation cycle (interval_counter).
  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.

- When the TA_PHS attribute is not specified
  The target cyclic handler activation timing is set up according to Activation cycle (interval_counter)) on the basis of the call time of this service call.
  This setting is performed regardless of the operating status of the target cyclic handler.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *cycid* ≤ 0 - *cycid* > VTMAX_CYH

Macro	Value	Description
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”. Note When the ista_cyc is issued from task or the sta_cyc is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.

stp_cyc
istp_cyc

Outline

Stop cyclic handler operation.

C format

```
ER      stp_cyc (ID cycid);
ER      istp_cyc (ID cycid);
```

Parameter(s)

I/O	Parameter	Description
I	`ID cycid;`	ID number of the cyclic handler.

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 RI600V4 until issuance of sta_cyc or ista_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.
E_ID	-18	Invalid ID number. - *cycid* ≤ 0 - *cycid* > VTMAX_CYH
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”. Note When the istp_cyc is issued from task or the stp_cyc is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.

ref_cyc
iref_cyc

Outline

Reference cyclic handler state.

C format

```
ER      ref_cyc (ID cycid, T_RCYC *pk_rcyc);
ER      iref_cyc (ID cycid, T_RCYC *pk_rcyc);
```

Parameter(s)

I/O	Parameter	Description
I	`ID cycid;`	ID number of the cyclic handler.
O	`T_RCYC *pk_rcyc;`	Pointer to the packet returning the cyclic handler state.

[Cyclic handler state packet: T_RCYC]

```
typedef struct  t_rcyc {
    STAT    cycstat;        /*Current state*/
    RELTIM  lefttim;        /*Time left before the next activation*/
} T_RCYC;
```

Explanation

Stores cyclic handler state packet (current state, time until the next activation, etc.) of the cyclic handler specified by parameter *cycid* in the area specified by parameter *pk_rcyc*.

- *cycstat*
  Store the current state.

  TCYC_STP: Non-operational state
  TCYC_STA: Operational state

- *lefttim*
  Stores the time until the next activation (in millisecond). When the target cyclic handler is in the non-operational state, lefttim is undefined.

Return value

Macro	Value	Description
E_OK	0	Normal completion.

Macro	Value	Description
E_ID	-18	Invalid ID number. - *cycid* ≤ 0 - *cycid* > VTMAX_CYH
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”. Note When the iref_cyc is issued from task or the ref_cyc is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.

sta_alm
ista_alm

Outline

Start alarm handler operation.

C format

```
ER      sta_alm (ID almid, RELTIM almtim);
ER      ista_alm (ID almid, RELTIM almtim);
```

Parameter(s)

I/O	Parameter	Description
I	`ID almid;`	ID number of the alarm handler.
I	`RELTIM almtim;`	Activation time (unit: msec)

Explanation

This service call sets to start the alarm handler specified by parameter *almid* in *almtim* msec and moves the target alarm handler from the non-operational state (STP state) to operational state (STA state).
As a result, the target alarm handler is handled as an activation target of the RI600V4.

Note 1 When 0 is specified for almtim, the alarm handler will start at next base clock interrupt.

Note 2 This service call sets the activation time even if the target alarm handler has already been in the operational state. The previous activation time becomes invalid.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *almtim* > (0x7FFFFFFF - TIC_NUME) / TIC_DENO
E_ID	-18	Invalid ID number. - *almid* ≤ 0 - *almid* > VTMAX_ALH

Macro	Value	Description
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”. Note When the ista_alm is issued from task or the sta_alm is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.

stp_alm
istp_alm

Outline

Stop alarm handler operation.

C format

```
ER      stp_alm (ID almid);
ER      istp_alm (ID almid);
```

Parameter(s)

I/O	Parameter	Description
I	`ID almid;`	ID number of the alarm handler.

Explanation

This service call moves the alarm handler specified by parameter *almid* from the operational state (STA state) to non-operational state (STP state).
As a result, the target alarm handler is excluded from activation targets of the RI600V4 until issuance of sta_alm or ista_alm.

Note This service call does not perform queuing of stop requests. If the target alarm 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.
E_ID	-18	Invalid ID number. - *almid* ≤ 0 - *almid* > VTMAX_ALH
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”. Note When the istp_alm is issued from task or the stp_alm is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.

ref_alm
iref_alm

Outline

Reference alarm handler state.

C format

```
ER      ref_alm (ID almid, T_RALM *pk_ralm);
ER      iref_alm (ID almid, T_RALM *pk_ralm);
```

Parameter(s)

I/O	Parameter	Description
I	`ID almid;`	ID number of the alarm handler.
O	`T_RALM *pk_ralm;`	Pointer to the packet returning the alarm handler state.

[Alarm handler state packet: T_RALM]

```
typedef struct  t_ralm {
    STAT    almstat;        /*Current state*/
    RELTIM  lefttim;        /*Time left before the next activation*/
} T_RALM;
```

Explanation

Stores alarm handler state packet (current state, time until the next activation, etc.) of the alarm handler specified by parameter *almid* in the area specified by parameter *pk_ralm*.

- *almstat*
  Store the current state.

  TALM_STP: Non-operational state
  TALM_STA: Operational state

- *lefttim*
  Stores the time until the next activation (in millisecond). When the target alarm handler is in the non-operational state, lefttim is undefined.

Return value

Macro	Value	Description
E_OK	0	Normal completion.

Macro	Value	Description
E_ID	-18	Invalid ID number. - *almid* ≤ 0 - *almid* > VTMAX_ALH
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”. Note When the iref_alm is issued from task or the ref_alm is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.

18.2.12 System state management functions

The following shows the service calls provided by the RI600V4 as the system state management functions.

Table 18-14 System State Management Functions

Service Call	Function	Useful Range
rot_rdq	Rotate task precedence	Task
irot_rdq	Rotate task precedence	Non-task
get_tid	Reference task ID in the RUNNING state	Task
iget_tid	Reference task ID in the RUNNING state	Non-task
loc_cpu	Lock the CPU	Task
iloc_cpu	Lock the CPU	Non-task
unl_cpu	Unlock the CPU	Task
iunl_cpu	Unlock the CPU	Non-task
dis_dsp	Disable dispatching	Task
ena_dsp	Enable dispatching	Task
sns_ctx	Reference contexts	Task, Non-task
sns_loc	Reference CPU locked state	Task, Non-task
sns_dsp	Reference dispatching disabled state	Task, Non-task
sns_dpn	Reference dispatch pending state	Task, Non-task
vsys_dwn	System down	Task, Non-task
ivsys_dwn	System down	Task, Non-task
vsta_knl	Start RI600V4	Task, Non-task
ivsta_knl	Start RI600V4	Task, Non-task

rot_rdq
irot_rdq

Outline

Rotate task precedence.

C format

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

Parameter(s)

I/O	Parameter	Description
I	PRI *tskpri;*	Priority of the tasks. TPRI_SELF: Current priority of the invoking task. Value: Priority of the tasks.

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 RI600V4'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.

Note 4 As for a task which has locked mutexes, the current priority might be different from the base priority. In this case, even if the task issues this servie call specifying TPRI_SELF for parameter *tskpri*, the ready queue of the current priority that the invoking task belongs cannot be changed.

Note 5 For current priority and base priority, refer to "6.2.2 Current priority and base priority".

Return value

Macro	Value	Description
E_OK	0	Normal completion.

Macro	Value	Description
E_PAR	-17	Parameter error. - *tskpri* < 0 - *tskpri* > TMAX_TPRI - When this service call was issued from a non-task, TPRI_SELF was specified *tskpri*.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - The irot_rdq was issued from task. - The rot_rdq was issued from non-task. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.

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);
```

Parameter(s)

I/O	Parameter	Description
O	`ID *p_tskid;`	Pointer to the area returning the task ID number.

Explanation

These service calls store the ID of a task in the RUNNING state in the area specified by parameter *p_tskid*.
This service call stores TSK_NONE in the area specified by parameter *p_tskid* if no tasks that have entered the RUNNING state exist.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status "PSW.IPL > kernel interrupt mask level". Note When the iget_tid is issued from task or the get_tid is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.

loc_cpu
iloc_cpu

Outline

Lock the CPU.

C format

```
ER      loc_cpu (void);
ER      iloc_cpu (void);
```

Parameter(s)

None.

Explanation

These service calls transit the system to the CPU locked state.
In the CPU locked state, the task scheduling is prohibited, and kernel interrupts are masked. Therefore, exclusive processing can be achieved for all processing programs except non-kernel interrupt handlers.
The service calls that can be issued in the CPU locked state are limited to the one listed below.

Service Call that can be issued	Function
ext_tsk	Terminate invoking task. (This service call transit the system to the CPU unlocked state.)
loc_cpu, iloc_cpu	Lock the CPU.
unl_cpu, iunl_cpu	Unlock the CPU.
sns_loc	Reference CPU state.
sns_dsp	Reference dispatching state.
sns_ctx	Reference contexts.
sns_dpn	Reference dispatch pending state.
vsys_dwn, ivsys_dwn	System down

The unl_cpu, iunl_cpu and ext_tsk releases from the CPU locked state,

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

Note 2 These service calls do 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 RI600V4 realizes the TIME MANAGEMENT FUNCTIONS by using base clock timer interrupts that occurs at constant intervals. If acknowledgment of the relevant base clock timer interrupt is disabled by issuing this service call, the TIME MANAGEMENT FUNCTIONS may no longer operate normally.

Note 4 For kernel interrupts, refer to "10.1 Interrupt Type".

Note 5 The loc_cpu returns E_ILUSE error while interrupt mask has changed to other than 0 by chg_ims.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_CTX	-25	Context error. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”. Note When the iloc_cpu is issued from task or the loc_cpu is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.
E_ILUSE	-28	Illegal use of service call. - This service call is issued in the status that the invoking task changes the PSW.IPL to other than 0 by using chg_ims.

unl_cpu
iunl_cpu

Outline

Unlock the CPU.

C format

```
ER      unl_cpu (void);
ER      iunl_cpu (void);
```

Parameter(s)

None.

Explanation

These service calls transit the system to the CPU unlocked state.

Note 1 These service calls do 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 These service calls do not cancel the dispatching disabled state that was set by issuing dis_dsp.

Note 3 The CPU locked state is also cancelled by ext_tsk.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_CTX	-25	Context error. - The iunl_cpu was issued from task. - The unl_cpu was issued from task. - This service call was issued in the status "PSW.IPL > kernel interrupt mask level".

dis_dsp

Outline

Disable dispatching.

C format

```
ER      dis_dsp (void);
```

Parameter(s)

None.

Explanation

This service call transits the system to the dispatching disabled state.
In the dispatching disabled state, the task scheduling is prohibited. Therefore, exclusive processing can be achieved for all tasks.
The operation that transit the system to the dispatching disabled state is as follows.

- dis_dsp
- chg_ims that changes PSW.IPL to other than 0.

The operation that transit the system to the dispatching enabled state is as follows.

- ena_dsp
- ext_tsk
- chg_ims that changes PSW.IPL to 0.

Note 1 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.

Note 2 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 3 If a service call (such as wai_sem, wai_flg) that may move the status of the invoking task is issued while the dispatching disabled state, that service call returns E_CTX regardless of whether the required condition is immediately satisfied.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.

ena_dsp

Outline

Enable dispatching.

C format

```
ER      ena_dsp (void);
```

Parameter(s)

None.

Explanation

This service call transits the system to the dispatching enabled state.
The operation that changes in the dispatching disabled state is as follows.

- dis_dsp
- chg_ims that changes PSW.IPL to other than 0.

The operation that changes in the dispatching enabled state is as follows.

- ena_dsp
- ext_tsk
- chg_ims that changes PSW.IPL to 0.

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

Note 2 If a service call (such as wai_sem, wai_flg) that may move the status of the invoking task is issued from when dis_dsp is issued until this service call is issued, the RI600V4 returns E_CTX regardless of whether the required condition is immediately satisfied.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the status "PSW.IPL > kernel interrupt mask level".

sns_ctx

Outline

Reference contexts.

C format

```
BOOL     sns_ctx (void);
```

Parameter(s)

None.

Explanation

This service call examines the context type of the processing program that issues this service call. This service call returns TRUE when the processing program is non-task context, and return FALSE when the processing program is task context.

Return value

Macro	Value	Description
TRUE	1	Normal completion (non-task context).
FALSE	0	Normal completion (task context).
E_CTX	-25	Context error. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.

sns_loc

Outline

Reference CPU locked state.

C format

```
BOOL    sns_loc (void);
```

Parameter(s)

None.

Explanation

This service call examines whether the system is in the CPU locked state or not. This service call returns TRUE when the system is in the CPU locked state, and return FALSE when the system is in the CPU unlocked state.

Return value

Macro	Value	Description
TRUE	1	Normal completion (CPU locked state).
FALSE	0	Normal completion (CPU unlocked state).
E_CTX	-25	Context error. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.

sns_dsp

Outline

Reference dispatching disabled state.

C format

```
BOOL        sns_dsp (void);
```

Parameter(s)

None.

Explanation

This service call examines whether the system is in the dispatching disabled state or not. This service call returns TRUE when the system is in the dispatching disabled state, and return FALSE when the system is in the dispatching enabled state.

Return value

Macro	Value	Description
TRUE	1	Normal completion (dispatching disabled state).
FALSE	0	Normal completion (dispatching enabled state).
E_CTX	-25	Context error. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.

sns_dpn

Outline

Reference dispatch pending state.

C format

```
BOOL    sns_dpn (void);
```

Parameter(s)

None.

Explanation

This service call examines whether the system is in the dispatch pending state or not. This service call returns TRUE when the system is in the dispatch pending state, and return FALSE when the system is not in the dispatch pending state.
The state to fill either the following is called dispatch pending state.

- Dispatching disabled state
- CPU locked state
- PSW.IPL > 0, such as handlers

Return value

Macro	Value	Description
TRUE	1	Normal completion. (dispatch pending state)
FALSE	0	Normal completion. (any other states)
E_CTX	-25	Context error. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.

vsys_dwn
ivsys_dwn

Outline

System down.

C format

```
void    vsys_dwn(W type, VW inf1, VW inf2, VW inf3);
void    vsys_dwn(W type, VW inf1, VW inf2, VW inf3);
```

Parameter(s)

I/O	Parameter	Description
I	`W type;`	Error type.
I	`VW inf1;`	System down information 1
I	`VW inf2;`	System down information 2
I	`VW inf3;`	System down information 3

Explanation

These service calls pass the control to the System down routine (_RI_sys_dwn__).
Specify the value (from 1 to 0x7FFFFFFF) typed to the occurring error for *type*. Note the value of 0 or less is reserved by the RI600V4.
These service calls never return.
For details of the parameter specification, refer to "13.2.2 Parameters of system down routine".
These service calls are the function outside the range of μITRON4.0 specifications.

Note The system down routine is also called when abnormality is detected in the RI600V4.

Return value

None.

vsta_knl
ivsta_knl

Outline

Start RI600V4.

C format

```
void    vsta_knl( void );
void    vsta_knl( void );
```

Parameter(s)

None.

Explanation

These service start the RI600V4.
These service calls never return.
When these service call is issued, it is necessary to fill the following.

- All interrupts can not be accepted. (For example, PSW.I == 0)
- The CPU is in the supervisor mode (PSW.PM == 0).

The outline of processing of these service calls is shown as follows.

1) Initialize ISP register to the end address of SI section + 1
2) Initialize INTB register to the start address of the relocatable vector table (INTERRUPT_VECTOR section). The relocatable vector table is generated by the cfg600.
3) Initialize the system time to 0.
4) Create various object which are defined in the system configuration file.
5) Pass control to scheduler

These service calls are the function outside the range of μITRON4.0 specifications.

Return value

None.

18.2.13 Interrupt management functions

The following shows the service calls provided by the RI600V4 as the interrupt management functions.

Table 18-15 Interrupt Management Functions

Service Call	Function	Useful Range
chg_ims	Change interrupt mask	Task
ichg_ims	Change interrupt mask	Non-task
get_ims	Reference interrupt mask	Task
iget_ims	Reference interrupt mask	Non-task

chg_ims
ichg_ims

Outline

Change interrupt mask.

C format

```
ER      chg_ims (IMASK imask);
ER      ichg_ims (IMASK imask);
```

Parameter(s)

I/O	Parameter	Description
I	IMASK *imask*;	Interrupt mask desired.

Explanation

These service calls change PSW.IPL to the value specified by *imask*. Ranges of the value that can be specified for *imask* are from 0 to 15.
In the chg_ims, the system shifts to the dispatching disabled state when other than 0 is specified for *imask*, (it is equivalent to dis_dsp.) and shifts to the dispatching enabled state when 0 is specified for *imask* (it is equivalent to ena_dsp.).
On the other hand, the ichg_ims does not change the dispatching disabled / enabled state.
The service calls that can be issued while PSW.IPL is larger than the Kernel interrupt mask level (system_IPL) are limited to the one listed below.

Service Call that can be issued	Function
chg_ims, ichg_ims	Change interrupt mask.
get_ims, iget_ims	Reference interrupt mask
vsys_dwn, ivsys_dwn	System down
vsta_knl, ivsta_knl	Start RI600V4.

Note 1 In the non-task, the interrupt mask must not lower PSW.IPL more than it starts.

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

Note 3 If a service call (such as wai_sem, wai_flg) that may move the status of the invoking task is issued while the dispatching disabled state, that service call returns E_CTX regardless of whether the required condition is immediately satisfied.

Note 4 The RI600V4 realizes the TIME MANAGEMENT FUNCTIONS by using base clock timer interrupts that occurs at constant intervals. If acknowledgment of the relevant base clock timer interrupt is disabled by issuing this service call, the TIME MANAGEMENT FUNCTIONS may no longer operate normally.

Note 5 Do not issue ena_dsp while a task changes PSW.IPL to other than 0 by using chg_ims. If issuing ena_dsp, the system moves to the dispatching enabled state. If task dispatching occurs, PSW is changed for the dispatched task. Therefore PSW.IPL may be lowered without intending it

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *imask* > 15
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - The ichg_ims was issued from task. - The chg_ims was issued from non-task.

get_ims
iget_ims

Outline

Reference interrupt mask.

C format

```
ER      get_ims (IMASK *p_imask);
ER      iget_ims (IMASK *p_imask);
```

Parameter(s)

I/O	Parameter	Description
O	`IMASK *p_imask;`	Pointer to the area returning the interrupt mask.

Explanation

These service calls store PSW.IPL into the area specified by parameter *p_imask*.

Note 1 These service call do not detect the context error.

Note 2 The following intrinsic functions provided by compiler are higher-speed than this service call. See “CubeSuite+ Integrated Development Environment User's Manual: RX Coding” for details about intrinsic functions.

- get_ipl() : Refers to the interrupt priority level.
- get_psw() : Refers to PSW value.

Return value

Macro	Value	Description
E_OK	0	Normal completion.

18.2.14 System configuration management functions

The following shows the service calls provided by the RI600V4 as the system configuration management functions.

Table 18-16 System Configuration Management Functions

Service Call	Function	Useful Range
ref_ver	Reference version information	Task
iref_ver	Reference version information	Non-task

ref_ver
iref_ver

Outline

Reference version information.

C format

```
ER      ref_ver (T_RVER *pk_rver);
ER      iref_ver (T_RVER *pk_rver;
```

Parameter(s)

I/O	Parameter	Description
O	`T_RVER *pk_rver;`	Pointer to the packet returning the version information.

[Version information packet: T_RVER]

```
typedef struct  t_rver {
    UH      maker;          /*Kernel maker 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*/
} T_RVER;
```

Explanation

These service calls store the RI600V4 version information into the area specified by parameter pk_rver.

- *maker*
  The *maker* represents the manufacturer who created this kernel. In the RI600V4, 0x011B, which is the maker code assigned for Renesas Electronics Corporation, is returned for maker.
  Note, the value defined in the kernel configuration macro TKERNEL_MAKER is same as *maker*.

- *prid*
  The *prid* represents the number that identifies the kernel and VLSI. In the RI600V4, 0x0003 is returned for *prid*.
  Note, the value defined in the kernel configuration macro TKERNEL_PRID is same as *prid*.

- *spver*
  The *spver* represents the specification to which this kernel conforms. In the RI600V4, 0x5403 is returned for *spver*.
  Note, the value defined in the kernel configuration macro TKERNEL_SPVER is same as *spver*.

- *prver*
  The *prver* represents the version number of this kernel.
  For example, 0x0123 is returned for *prver* when the kernel version is “V1.02.03”.
  Note, the value defined in the kernel configuration macro TKERNEL_PRVER is same as *prver*.

- *prno*
  The *prno* represents product management information and product number, etc. In the RI600V4, 0x0000 is returned for all *prnos*.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”. Note When the iref_ver is issued from task or the ref_ver is issued from non-task, the context error is not detected and normal operation of the system is not guaranteed.

18.2.15 Object reset functions

The following shows the service calls provided by the RI600V4 as the object reset functions.

Table 18-17 Object Reset Functions

Service Call	Function	Useful Range
vrst_dtq	Reset data queue	Task
vrst_mbx	Reset mailbox	Task
vrst_mbf	Reset message buffer	Task
vrst_mpf	Reset fixed-sized memory pool	Task
vrst_mpl	Reset variable-sized memory pool	Task

vrst_dtq

Outline

Reset data queue.

C format

```
ER      vrst_dtq (ID dtqid);
```

Parameter(s)

I/O	Parameter	Description
I	`ID dtqid;`	ID number of the data queue.

Explanation

This service call reset the data queue specified by parameter *dtqid*.
The data having been accumulated by the data queue area are annulled. The tasks to wait to send data to the target data queue are released from the WAITING state, and EV_RST is returned as a return value for the tasks.

Note 1 In this service call, the tasks to wait to receive data do not released from the WAITING state.

Note 2 This service call is the function outside μITRON4.0 specification.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *dtqid* ≤ 0 - *dtqid* > VTMAX_DTQ
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the status "PSW.IPL > kernel interrupt mask level".

vrst_mbx

Outline

Reset mailbox.

C format

```
ER      vrst_mbx (ID mbxid);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mbxid;`	ID number of the mailbox.

Explanation

This service call reset the mailbox specified by parameter *mbxid*.
The messages having been accumulated by the mailbox come off from the management of the RI600V4.

Note 1 In this service call, the tasks to wait to receive message do not released from the WAITING state.

Note 2 This service call is the function outside μITRON4.0 specification.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *mbxid* ≤ 0 - *mbxid* > VTMAX_MBX
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.

vrst_mbf

Outline

Reset message buffer.

C format

```
ER      vrst_mbf (ID mbfid);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mbfid;`	ID number of the message buffer.

Explanation

This service call reset the message buffer specified by parameter *mbfid*.
The messages having been accumulated by the message buffer area are annulled. The tasks to wait to send message to the target message buffer are released from the WAITING state, and EV_RST is returned as a return value for the tasks.

Note 1 In this service call, the tasks to wait to receive message do not released from the WAITING state.

Note 2 This service call is the function outside μITRON4.0 specification.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *mbfid* ≤ 0 - *mbfid* > VTMAX_MBF
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.

vrst_mpf

Outline

Reset fixed-sized memory pool.

C format

```
ER      vrst_mpf (ID mpfid);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mpfid;`	ID number of the fixed-sized memory pool.

Explanation

This service call reset the fixed-sized memory pool specified by parameter *mpfid*.
The tasks to wait to get memory block from the target fixed-sized memory pool are released from the WAITING state, and EV_RST is returned as a return value for the tasks.

Note 1 All fixed-sized memory blocks that had already been acquired are returned to the target fixed-sized memory pool. Therefore, do not access those fixed-sized memory blocks after issuing this service call.

Note 2 This service call is the function outside μITRON4.0 specification.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *mpfid* ≤ 0 - *mpfid* > VTMAX_MPF
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the status “PSW.IPL > kernel interrupt mask level”.

vrst_mpl

Outline

Reset variable-sized memory pool.

C format

```
ER      vrst_mpl (ID mplid);
```

Parameter(s)

I/O	Parameter	Description
I	ID *mplid*;	ID number of the variable-sized memory pool.

Explanation

This service call reset the variable-sized memory pool specified by parameter *mplid*.
The tasks to wait to get memory block from the target variable-sized memory pool are released from the WAITING state, and EV_RST is returned as a return value for the tasks.

Note 1 All variable-sized memory blocks that had already been acquired are returned to the target variable-sized memory pool. Therefore, do not access those variable-sized memory blocks after issuing this service call.

Note 2 This service call is the function outside μITRON4.0 specification.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *mplid* ≤ 0 - *mplid* > VTMAX_MPL
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the status "PSW.IPL > kernel interrupt mask level".

CHAPTER 19 SYSTEM CONFIGURATION FILE

This chapter explains the coding method of the system configuration file required to output information files that contain data to be provided for the RI600V4.

19.1 Outline

The following shows the notation method of system configuration files.

- Comment
  Parts from two successive slashes (//) to the line end are regarded as comments.

- Numeric
  A numeric value can be written in one of the following formats. Note, do not specify the value exceeding 0xFFFFFFFF.

  Hexadecimal: Add “0x” or “0X” at the beginning of a numeric value or add “h” or “H” at the end. In the latter format, be sure to add “0” at the beginning when the value begins with an alphabetic letter from A to F or a to f. Note that the configurator does not distinguish between uppercase and lowercase letters for alphabetic letters (A to F or a to f) used in numeric value representation.
  Decimal: Simply write an integer value as is usually done (23, for example). Note that a decimal value must not begin with “0”.
  Octal: Add “0” at the beginning of a numeric value or add “O” or “o” at the end.
  Binary: Add “B” or “b” at the end of a numeric value. Note that a binary value must not begin with “0”.

- Operator
  The following operator can be used for numeric value.

Table 19-1 Operator

Operator	Precedence	Direction of Computation
( )	High	Left to right
- (unary minus)		Right to left
* / %		Left to right
+ - (binary minus)	Low	Left to right

- Symbol
  A symbol is a string of numeric characters, uppercase alphabetic letters, lowercase alphabetic letters, and underscores (_). It must not begin with a numeric character.

- Function name
  A function name consists of numeric characters, uppercase alphabetic letters, lowercase alphabetic letters, underscores (_), and dollar signs ($). It must not begin with a numeric character and must end with “()”.
  To specify module name written by assembly language, name the module starting in '_', and specify the name that excludes '_' for function name.

- Frequency
  The frequency is indicated by a character string that consist of numerals and . (period), and ends with “MHz”. The numerical values are significant up to six decimal places. Also note that the frequency can be entered using

19.2 Default System Configuration File

For most definition items, if the user omits settings, the settings in the default system configuration file are used. The default system configuration file is stored in the folder indicated by environment variable “LIB600”. Be sure not to edit this file.

19.3 Configuration Information (static API)

The configuration information that is described in a system configuration file is shown as follows.

- System Information (system)
- Base Clock Interrupt Information (clock)
- Task Information (task[])
- Semaphore Information (semaphore[])
- Eventflag Information (flag[])
- Data Queue Information (dataqueue[])
- Mailbox Information (mailbox[])
- Mutex Information (mutex[])
- Message Buffer Information (message_buffer[])
- Fixed-sized Memory Pool Information (memorypool[])
- Variable-sized Memory Pool Information (variable_memorypool[])
- Cyclic Handler Information (cyclic_hand[])
- Alarm Handler Information (alarm_handl[])
- Relocatable Vector Information (interrupt_vector[])
- Fixed Vector/Exception Vector Information (interrupt_fvector[])

19.4 System Information (system)

Here, information on the system whole is defined.
Only one “system” can be defined. And the “system” can not be omitted.

Format

Parentheses < >show the user input part.

```
system {
    stack_size  = <1. System stack size (stack_size)>;
    priority    = <2. Maximum task priority (priority)>;
    system_IPL  = <3. Kernel interrupt mask level (system_IPL)>;
    message_pri = <4. Maximum message priority (message_pri)>;
    tic_deno    = <5. Denominator of base clock interval time (tic_deno)>;
    tic_nume    = <6. Numerator of base clock interval time (tic_nume)>;
    context     = <7. Task context register (context)>;
};
```

1) System stack size (*stack_size*)

- Description
  Define the total stack size used in service call processing and interrupt processing.

- Definition format
  Numeric value

- Definition range
  More than 8, and multiple of 4.

- When omitting
  The set value in the default system configuration file (factory setting: 0x800) applied.

2) Maximum task priority (*priority*)

- Description
  Define the maximum task priority.

- Definition format
  Numeric value

- Definition range
  1 - 255

- When omitting
  The set value in the default system configuration file (factory setting: 32) applied.

- TMAX_TPRI
  The cfg600 outputs the macro TMAX_TPRI which defines this setting to the system information header file “kernel_id.h”.

3) Kernel interrupt mask level (*system_IPL*)

- Description
  Define the interrupt mask level when the kernel's critical section is executed (PSW register's IPL value). Interrupts with higher priority levels than that are handled as “non-kernel interrupts”.
  For details of “non-kernel interrupts” and “kernel interrupts”, refer to “10.1 Interrupt Type”.

- Definition format
  Numeric value

- Definition range
  1 - 15

- When omitting
  The set value in the default system configuration file (factory setting: 7) applied.

- VTKNL_LVL
  The cfg600 outputs the macro VTKNL_LVL which defines this setting to the system information header file "kernel_id.h".

4) Maximum message priority (*message_pri*)

- Description
  Define the maximum message priority used in the mailbox function. Note that if the mailbox function is not used, this definition item has no effect.

- Definition format
  Numeric value

- Definition range
  1 - 255

- When omitting
  The set value in the default system configuration file (factory setting: 255) applied.

- TMAX_MPRI
  The cfg600 outputs the macro TMAX_MPRI which defines this setting to the system information header file "kernel_id.h".

5) Denominator of base clock interval time (*tic_deno*)

- Description
  The base clock interval time is calculated by the following expression. Either *tic_deno* or *tic_nume* should be 1.

  The base clock interval time (in millisecond) = *tic_nume* / *tic_deno*

- Definition format
  Numeric value

- Definition range
  1 - 100

- When omitting
  The set value in the default system configuration file (factory setting: 1) applied.

- TIC_DENO
  The cfg600 outputs the macro TIC_DENO which defines this setting to the system information header file "kernel_id.h".

6) Numerator of base clock interval time (*tic_nume*)

- Description
  See above.

- Definition format
  Numeric value

- Definition range
  1 - 65535

- When omitting
  The set value in the default system configuration file (factory setting: 1) applied.

- TIC_NUME
  The cfg600 outputs the macro TIC_NUME which defines this setting to the system information header file "kernel_id.h".

7) Task context register (*context*)

- Description
  Define the register set used by tasks. The settings made here apply to all tasks.
- Definition format
  Symbol
- Definition range
  Select one from item of “Setting” in Table 19-2.

Table 19-2 system.context

Setting	CPU		FPU	DSP
	PSW, PC, R0 - R7, R14, R15	R8 - R13	FPSW	Accumulator [a]
NO	Guaranteed	Guaranteed	Not guaranteed	Not guaranteed
FPSW	Guaranteed	Guaranteed	Guaranteed	Not guaranteed
ACC	Guaranteed	Guaranteed	Not guaranteed	Guaranteed
FPSW,ACC	Guaranteed	Guaranteed	Guaranteed	Guaranteed
MIN	Guaranteed	Not guaranteed	Not guaranteed	Not guaranteed
MIN,FPSW	Guaranteed	Not guaranteed	Guaranteed	Not guaranteed
MIN,ACC	Guaranteed	Not guaranteed	Not guaranteed	Guaranteed
MON,FPSW,ACC	Guaranteed	Not guaranteed	Guaranteed	Guaranteed

a. When compiler option “-isa=rxv2” is specified, the “Accumulator” means ACC0 register and ACC1 register. In the case of others, the “Accumulator” means ACC0 register (in RXv2 architecture) or ACC register (in RXV1 architecture).

Note Compiler option “-isa” is supported by the compiler CC-RX V2.01 or later.

- When omitting
  The set value in the default system configuration file (factory setting: NO) applied.
- Note
  Be sure to refer to “19.5 Note Concerning system.context”.

19.5 Note Concerning system.context

This sections explains note concerning system.context.

19.5.1 Note concerning FPU and DSP

The setting for system.context differs depending on how FPU and DSP are handled.
The recommendation setting of system.context is indicated from now on. If other than recommended setting is specified, the RI600V4 performance may be slightly deteriorated, compared to the recommended settings case.

1) When using MCU that incorporates FPU and DSP (accumulator)
Corresponding MCUs: RX600 series, etc.

2) When using MCU that does not incorporate FPU, but incorporates DSP (accumulator)
Corresponding MCUs: RX200 series, etc.

3) When using MCU that incorporates FPU, but does not incorporate DSP (accumulator)
Corresponding MCUs: MCUs that corresponds to this doesn't exist at the time of making of this manual.

4) When using MCU that incorporate neither FPU nor DSP (accumulator)
Corresponding MCUs: MCUs that corresponds to this doesn't exist at the time of making of this manual.

Note The compiler outputs floating-point arithmetic instructions only when the “-fpu” option is specified. If the “-chkfpu” option is specified in the assembler, the floating-point arithmetic instructions written in a program are detected as warning.
In no case does the compiler output the DSP function instructions. If the “-chkdsp” option is specified in the assembler, the DSP function instructions written in a program are detected as warning.

1) When using MCU that incorporates FPU and DSP (accumulator)

Table 19-3 When using MCU that incorporates FPU and DSP (accumulator)

Usage condition of instruction in tasks		Recommendation setting of system.context
Floating point arithmetic instructions	DSP function instructions	
YES	YES	“FPSW” and “ACC” included settings essential
	NO	“FPSW” included setting essential and “ACC” excluded setting recommended
NO	YES	“ACC” included setting essential and “FPSW” excluded setting recommended
	NO	“FPSW” and “ACC” excluded settings recommended

2) When using MCU that does not incorporate FPU, but incorporates DSP (accumulator)

Table 19-4 When using MCU that does not incorporate FPU, but incorporates DSP (accumulator)

Usage condition of instruction in tasks		Recommendation setting of system.context
Floating point arithmetic instructions	DSP function instructions	
YES	YES	Since the MCU does not incorporate FPU, floating-point arithmetic instructions cannot be used.
	NO	
NO	YES	“FPSW” excluded and “ACC” included settings essential
	NO	“FPSW” excluded setting essential and “ACC” excluded settings recommended

3) When using MCU that incorporates FPU, but does not incorporate DSP (accumulator)

Table 19-5 When using MCU that incorporates FPU, but does not incorporate DSP (accumulator)

Usage condition of instruction in tasks		Recommendation setting of system.context
Floating point arithmetic instructions	DSP function instructions	
YES	YES	Since the MCU does not incorporate DSP, DSP function instructions cannot be used.
	NO	“FPSW” included and “ACC” excluded settings essential
NO	YES	Since the MCU does not incorporate DSP, DSP function instructions cannot be used.
	NO	“ACC” excluded setting essential and “FPSW” excluded settings recommended

4) When using MCU that incorporate neither FPU nor DSP (accumulator)

Table 19-6 When using MCU that incorporate neither FPU nor DSP (accumulator)

Usage condition of instruction in tasks		Recommendation setting of system.context
Floating point arithmetic instructions	DSP function instructions	
YES	YES	Since the MCU incorporate neither FPU nor DSP, floating-point arithmetic instructions and DSP function instructions cannot be used.
	NO	
NO	YES	
	NO	“FPSW” and “ACC” excluded settings essential

19.5.2 Relationship with the compiler options “-fint_register”, “-base” and “-pid”

In system.context, by selecting one of choices “MIN,” “MIN, ACC”, “MIN, FPSW,” or “MIN, ACC, FPSW,” it is possible to configure the registers so that R8- R13 registers will not be saved as task context. This results in an increased processing speed.
Note, however, that such a setting of system.context is permitted in only the case where all of R8 - R13 registers are specified to be used by the compiler options “-fint_register”, “-base” and “-pid”.
If, in any other case, the above setting is made for system.context, the kernel will not operate normally.

- Good example:

  1) -fint_register=4 -base=rom=R8 -base=ram=R9

  2) -fint_register=3 -base=rom=R8 -base=ram=R9 -base=0x80000=R10

- Bad example:

  3) No “-fint_register”, “-base” and “-pid” options

  4) -fint_register=4

  5) -base=rom=R8 -base=ram=R9

  6) -fint_register=3 -base=rom=R8 -base=ram=R9

19.6 Base Clock Interrupt Information (clock)

Here, information on the base clock interrupt is defined. The cfg600 outputs the file “ri_cmt.h” where the base clock timer initialization function (void _RI_init_cmt(void)) is described.
Only one “clock” can be defined.

Format

Parentheses < >show the user input part.

```
clock {
    timer       = <1. Selection of timer channel for base clock (timer)>;
    template    = <2. Template file (template)>;
    timer_clock = <3. CMT frequency (timer_clock)>;
    IPL         = <4. Base clock interrupt priority level (IPL)>;
};
```

1) Selection of timer channel for base clock (*timer*)

- Description
  Define the timer channel for the base clock.
- Definition format
  Symbol
- Definition range
  Select one from Table 19-7.

Table 19-7 clock.timer

Setting	Description
CMT0	Use CMT channel 0 assigned to relocatable vector 28.
CMT1	Use CMT channel 1 assigned to relocatable vector 29.
CMT2	Use CMT channel 2 assigned to relocatable vector 30.
CMT3	Use CMT channel 3 assigned to relocatable vector 31.
OTHER	Use a timer other than the above. In this case, the user needs to create a timer initialize routine.
NOTIMER	Do not use the base clock interrupt.

Note 1 The CMT (Compare Match Timer) is the timer that is mounted on RX MCU typically.

Note 2 Do not select “CMT2” and “CMT3” when CMT channel 2 and channel 3 are not mounted with RX MCU to use, and when relocatable vector assigned to CMT channel 2 and channel 3 is different from Table 19-7 with RX MCU to use.
For example, RX111 does not support CMT channel 2 and channel 3. And in RX64M, relocatable vector assigned to CMT channel 2 and channel 3 is not 30 and 31.

- When omitting
  The set value in the default system configuration file (factory setting: “CMT0”) applied.

2) Template file (*template*)

- Description
  Specify template file where hardware information and initialization function of CMT is described.
  This definition is ignored when either “NOTIMER” or “OTHER” is specified for *timer*.
  The template files are provided by the RI600V4. The template files may be added in the future version.
  Refer to the release notes for MCUs supported by each template file.
  Either CMT1, CMT2 or CMT3 might be unsupported according to template file. When the unsupported CMT channel is specified for *timer*, the cfg600 does not detect error but the error is detected at compilation of the file which includes “ri_cmt.h”.

- Definition format
  Symbol

- Definition range
  -

- When omitting
  The set value in the default system configuration file (factory setting: “rx610.tpl”) applied.

3) CMT frequency (*timer_clock*)

- Description
  Define frequency of the clock supplied to CMT. Please specify the frequency of PCLK (peripheral clock).

- Definition format
  Frequency

- Definition range
  -

- When omitting
  The set value in the default system configuration file (factory setting: “25MHz”) applied.

4) Base clock interrupt priority level (*IPL*)

- Description
  Define the interrupt priority level of the base clock interrupt.

- Definition format
  Numeric value

- Definition range
  From 1 to Kernel interrupt mask level (system_IPL) in System Information (system)

- When omitting
  The set value in the default system configuration file (factory setting: 4) applied.

- VTIM_LVL
  The cfg600 outputs the macro VTIM_LVL which defines this setting to the system information header file “kernel_id.h”.

19.7 Task Information (task[])

Here, each task is defined.

Format

Parentheses < >show the user input part.

```
task[ <1. ID number> ] {
    name          = <2. ID name (name)>;
    entry_address = <3. Task entry address (entry_addreess)>;
    stack_size    = <4. User stack size (stack_size)>;
    stack_section = <5. Section name assigned to the stack area (stack_section)>;
    priority      = <6. Task initial priority (priority)>;
    initial_start = <7. TA_ACT attribute (initial_start)>;
    exinf         = <8. Extended information (exinf)>;
};
```

1) ID number

- Description
  Define the task ID number.
- Definition format
  Numeric value
- Definition range
  From 1 to 255
- When omitting
  The cfg600 assigns the ID number automatically.
- Note
  The ID numbers must be assigned without an omission beginning with 1. Therefore, when specifying an ID number, be sure that the specified value is equal to or less than the number of objects defined.

2) ID name (*name*)

- Description
  Define the ID name. The specified ID name is output to the system information header file (kernel_id.h) in the form of the following.

```
#define    <ID name>    <ID number>
```

- Definition format
  Symbol
- Definition range
  -
- When omitting
  Cannot be omitted.

3) Task entry address (*entry_addreess*)

- Description
  Define the starting function of the task.
- Definition format
  Symbol
- Definition range
  -

- When omitting
  Cannot be omitted.

4) User stack size (*stack_size*)

- Description
  Define the user stack size.
- Definition format
  Numeric value
- Definition range
  More than the following values.

Table 19-8 Lower Bound Value of User Stack Size

Setting of system.context	Compiler option “-isa”	Lower bound value
NO	-	68
FPSW	-	72
ACC	“-isa=rxv2”	92
	“-isa=rxv1” or not specify “-isa”	76
FPSW,ACC	“-isa=rxv2”	96
	“-isa=rxv1” or not specify “-isa”	80
MIN	-	44
MIN,FPSW	-	48
MIN,ACC	“-isa=rxv2”	68
	“-isa=rxv1” or not specify “-isa”	52
MON,FPSW,ACC	“-isa=rxv2”	72
	“-isa=rxv1” or not specify “-isa”	56

Note Compiler option “-isa” is supported by the compiler CC-RX V2.01 or later.

- When omitting
  The set value in the default system configuration file (factory setting: 256) applied.

5) Section name assigned to the stack area (*stack_section*)

- Description
  Define the section name to be assigned to the user stack area.
  The cfg600 generates the user stack area with the size specified by *stack_size* to the section specified by *stack_section*. The section attribute is “DATA”, and the alignment number is 4.
  When linking, be sure to locate this section in the RAM area. Note, this section must not be located to address 0.
- Definition format
  Symbol
- Definition range
  -
- When omitting
  The set value in the default system configuration file (factory setting: “SURI_STACK”) applied.

6) Task initial priority (*priority*)

- Description
  Define the task initial priority.

- Definition format
  Numeric value

- Definition range
  From 1 to Maximum task priority (priority) in System Information (system)

- When omitting
  The set value in the default system configuration file (factory setting: 1) applied.

7) TA_ACT attribute (*initial_start*)

- Description
  Define the initial state of the task.

- Definition format
  Symbol

- Definition range
  Select either of the following:

ON:	Specify the TA_ACT attribute. (The initial state is READY state.)
OFF:	Not Specify the TA_ACT attribute. (The initial state is DORMANGT state.)

- When omitting
  The set value in the default system configuration file (factory setting: “OFF”) applied.

8) Extended information (*exinf*)

- Description
  Define the extended information of the task.

- Definition format
  Numeric value

- Definition range
  From 0 to 0xFFFFFFFF

- When omitting
  The set value in the default system configuration file (factory setting: 0) applied.

- Note
  When the task is activated by the TA_ACT attribute, act_tsk or iact_tsk, the extended information is passed to the task.

19.8 Semaphore Information (semaphore[])

Here, each semaphore is defined.

Format

Parentheses < >show the user input part.

```
semaphore[ <1. ID number> ] {
    name          = <2. ID name (name)>;
    max_count     = <3. Maximum resource count (max_count)>;
    initial_count = <4. Initial resource count (initial_count)>;
    wait_queue    = <5. Wait queue attribute (wait_queue)>;
};
```

1) ID number

- Description
  Define the semaphore ID number.

- Definition format
  Numeric value

- Definition range
  From 1 to 255

- When omitting
  The cfg600 assigns the ID number automatically.

- Note
  The ID numbers must be assigned without an omission beginning with 1. Therefore, when specifying an ID number, be sure that the specified value is equal to or less than the number of objects defined.

2) ID name (*name*)

- Description
  Define the ID name. The specified ID name is output to the system information header file (kernel_id.h) in the form of the following.

```
#define   <ID name>   <ID number>
```

- Definition format
  Symbol

- Definition range
  -

- When omitting
  Cannot be omitted.

3) Maximum resource count (*max_count*)

- Description
  Define the maximum resource count

- Definition format
  Numeric value

- Definition range
  From 1 to 65535

- When omitting
  The set value in the default system configuration file (factory setting: 1) applied.

4) Initial resource count (*initial_count*)

- Description
  Define the initial resource count.
- Definition format
  Numeric value
- Definition range
  From 0 to *max_count*
- When omitting
  The set value in the default system configuration file (factory setting: 1) applied.

5) Wait queue attribute (*wait_queue*)

- Description
  Define the wait queue attribute.
- Definition format
  Symbol
- Definition range
  Select either of the following:

  TA_TFIFO: FIFO order

  TA_TPRI: Task priority order
  Among tasks with the same priority, they are queued in FIFO order.
- When omitting
  The set value in the default system configuration file (factory setting: “TA_TFIFO”) applied.

19.9 Eventflag Information (flag[])

Here, each semaphore is defined.

Format

Parentheses < >show the user input part.

```
flag[ <1. ID number> ] {
    name            = <2. ID name (name)>;
    initial_pattern = <3. Initial bit pattern (initial_pattern)>;
    wait_multi      = <4. Multiple wait permission attribute (wait_multi)>;
    clear_attribute = <5. Clear attribute (clear_attribute)>;
    wait_queue      = <6. Wait queue attribute (wait_queue)>;
};
```

1) ID number

- Description
  Define the eventflag ID number.

- Definition format
  Numeric value

- Definition range
  From 1 to 255

- When omitting
  The cfg600 assigns the ID number automatically.

- Note
  The ID numbers must be assigned without an omission beginning with 1. Therefore, when specifying an ID number, be sure that the specified value is equal to or less than the number of objects defined.

2) ID name (*name*)

- Description
  Define the ID name. The specified ID name is output to the system information header file (kernel_id.h) in the form of the following.

```
#define   <ID name>   <ID number>
```

- Definition format
  Symbol

- Definition range
  \-

- When omitting
  Cannot be omitted.

3) Initial bit pattern (*initial_pattern*)

- Description
  Define the initial bit pattern

- Definition format
  Numeric value

- Definition range
  From 0 to 0xFFFFFFFF

- When omitting
  The set value in the default system configuration file (factory setting: 0) applied.

4) Multiple wait permission attribute (*wait_multi*)

- Description
  Define the attribute regarding whether multiple tasks are permitted to wait for the eventflag.

- Definition format
  Symbol

- Definition range
  Select either of the following:

  TA_WSGL: Not permit multiple tasks to wait for the eventflag.

  TA_WMUL: Permit multiple tasks to wait for the eventflag.

- When omitting
  The set value in the default system configuration file (factory setting: “TA_WSGL”) applied.

5) Clear attribute (*clear_attribute*)

- Description
  Define the clear attribute (TA_CLR).

- Definition format
  Symbol

- Definition range
  Select either of the following:

  NO: Not specify the TA_CLR attribute.

  YES: Specify the TA_CLR attribute.

- When omitting
  The set value in the default system configuration file (factory setting: “NO”) applied.

6) Wait queue attribute (*wait_queue*)

- Description
  Define the wait queue attribute.

- Definition format
  Symbol

- Definition range
  Select either of the following: However, when the TA_CLR attribute is not specified, the wait queue is managed in the FIFO order even if TA_TPRI is specified for *wait_queue*. This behavior falls outside μITRON4.0 specification.

  TA_TFIFO: FIFO order

  TA_TPRI: Task priority order
  Among tasks with the same priority, they are queued in FIFO order.

- When omitting
  The set value in the default system configuration file (factory setting: “TA_TFIFO”) applied.

19.10 Data Queue Information (dataqueue[])

Here, each data queue is defined.

Format

Parentheses < >show the user input part.

```
dataqueue[ <1. ID number> ] {
    name            = <2. ID name (name)>;
    buffer_size     = <3. Data count (buffer_size)>;
    wait_queue      = <4. Wait queue attribute (wait_queue)>;
};
```

1) ID number

- Description
  Define the data queue ID number.
- Definition format
  Numeric value
- Definition range
  From 1 to 255
- When omitting
  The cfg600 assigns the ID number automatically.
- Note
  The ID numbers must be assigned without an omission beginning with 1. Therefore, when specifying an ID number, be sure that the specified value is equal to or less than the number of objects defined.

2) ID name (*name*)

- Description
  Define the ID name. The specified ID name is output to the system information header file (kernel_id.h) in the form of the following.

```
#define   <ID name>   <ID number>
```

- Definition format
  Symbol
- Definition range
  -
- When omitting
  Cannot be omitted.

3) Data count (*buffer_size*)

- Description
  Define the number of data that the data queue can be stored.
- Definition format
  Numeric value
- Definition range
  From 0 to 65535
- When omitting
  The set value in the default system configuration file (factory setting: 0) applied.

4) Wait queue attribute (*wait_queue*)

- Description
  Define the wait queue attribute for sending.
  Note, task wait queue for receiving is managed in FIFO order.

- Definition format
  Symbol

- Definition range
  Select either of the following:

  TA_TFIFO: FIFO order

  TA_TPRI: Task current priority order
  Among tasks with the same current priority, they are queued in FIFO order.

- When omitting
  The set value in the default system configuration file (factory setting: “TA_TFIFO”) applied.

19.11 Mailbox Information (mailbox[])

Here, each mailbox is defined.

Format

Parentheses < >show the user input part.

```
mailbox[ <1. ID number> ] {
    name          = <2. ID name (name)>;
    wait_queue    = <3. Wait queue attribute (wait_queue)>;
    message_queue = <4. Message queue attribute (message_queue)>;
    max_pri       = <5. Maximum message priority (max_pri)>;
};
```

1) ID number

- Description
  Define the mailbox ID number.
- Definition format
  Numeric value
- Definition range
  From 1 to 255
- When omitting
  The cfg600 assigns the ID number automatically.
- Note
  The ID numbers must be assigned without an omission beginning with 1. Therefore, when specifying an ID number, be sure that the specified value is equal to or less than the number of objects defined.

2) ID name (*name*)

- Description
  Define the ID name. The specified ID name is output to the system information header file (kernel_id.h) in the form of the following.

```
#define    <ID name>    <ID number>
```

- Definition format
  Symbol
- Definition range
  -
- When omitting
  Cannot be omitted.

3) Wait queue attribute (*wait_queue*)

- Description
  Define the wait queue attribute.
- Definition format
  Symbol
- Definition range
  Select either of the following:

  TA_TFIFO: FIFO order

  TA_TPRI: Task priority order
  Among tasks with the same priority, they are queued in FIFO order.

- When omitting
  The set value in the default system configuration file (factory setting: “TA_TFIFO”) applied.

4) Message queue attribute (*message_queue*)

- Description
  Define the message queue attribute.
- Definition format
  Symbol
- Definition range
  Select either of the following:

  TA_MFIFO: The order of the message transmission request.

  TA_MPRI: Message priority order
- When omitting
  The set value in the default system configuration file (factory setting: “TA_MFIFO”) applied.

5) Maximum message priority (*max_pri*)

- Description
  When TA_MPRI is specified for *message_queue*, the message priority from 1 to *max_pri* can be used.
  When TA_MFIFO is specified for *message_queue*, this item is only disregarded.
- Definition format
  Numeric value
- Definition range
  From 1 to Maximum message priority (message_pri) in System Information (system)
- When omitting
  The set value in the default system configuration file (factory setting: 1) applied.

19.12 Mutex Information (mutex[])

Here, each mutex is defined.

Format

Parentheses < >show the user input part.

```
mutex[ <1. ID number> ] {
    name          = <2. ID name (name)>;
    ceilpri       = <3. Ceiling priority (ceilpri)>;
};
```

1) ID number

- Description
  Define the mutex ID number.
- Definition format
  Numeric value
- Definition range
  From 1 to 255
- When omitting
  The cfg600 assigns the ID number automatically.
- Note
  The ID numbers must be assigned without an omission beginning with 1. Therefore, when specifying an ID number, be sure that the specified value is equal to or less than the number of objects defined.

2) ID name (*name*)

- Description
  Define the ID name. The specified ID name is output to the system information header file (kernel_id.h) in the form of the following.

```
#define   <ID name>   <ID number>
```

- Definition format
  Symbol
- Definition range
  -
- When omitting
  Cannot be omitted.

3) Ceiling priority (*ceilpri*)

- Description
  The RI600V4 adopts Simplified priority ceiling protocol. The ceiling priority should be defined in *ceilpri*.
- Definition format
  Numeric value
- Definition range
  From 1 to Maximum task priority (priority) in System Information (system)
- When omitting
  The set value in the default system configuration file (factory setting: 1) applied.

19.13 Message Buffer Information (message_buffer[])

Here, each message buffer is defined.

Format

Parentheses < >show the user input part.

```
message_buffer[ <1. ID number> ] {
    name         = <2. ID name (name)>;
    mbf_size     = <3. Buffer size (mbf_size)>;
   mbf_section = <4. Section name assigned to the message buffer area (mbf_section)>;
    max_msgsz    = <5. Maximum message size (max_msgsz)>
};
```

1) ID number

- Description
  Define the message buffer ID number.
- Definition format
  Numeric value
- Definition range
  From 1 to 255
- When omitting
  The cfg600 assigns the ID number automatically.
- Note
  The ID numbers must be assigned without an omission beginning with 1. Therefore, when specifying an ID number, be sure that the specified value is equal to or less than the number of objects defined.

2) ID name (*name*)

- Description
  Define the ID name. The specified ID name is output to the system information header file (kernel_id.h) in the form of the following.

```
#define    <ID name>    <ID number>
```

- Definition format
  Symbol
- Definition range
  -
- When omitting
  Cannot be omitted.

3) Buffer size (*mbf_size*)

- Description
  Define the size of the message buffer in bytes.
- Definition format
  Numeric value
- Definition range
  0, or multiple of 4 in the range from 8 to 65532
- When omitting
  The set value in the default system configuration file (factory setting: 0) applied.

4) Section name assigned to the message buffer area (*mbf_section*)

- Description
  Define the section name to be assigned to the message buffer area.
  When *mbf_size* > 0, the cfg600 generates the message buffer area with the size specified by *buffer_size* to the section specified by *mbf_section*. The section attribute is “DATA”, and the alignment number is 4.
  When linking, be sure to locate this section in the RAM area. Note, this section must not be located to address 0.

- Definition format
  Symbol

- Definition range
  -

- When omitting
  The set value in the default system configuration file (factory setting: “BRI_HEAP”) applied.

5) Maximum message size (*max_msgsz*)

- Description
  Define the maximum message size of the message buffer in bytes.
  When *mbf_size* > 0, *max_msgsz* must be less than or equal to “*mbf_size* - 4”.

- Definition format
  Numeric value

- Definition range
  From 1 to 65528

- When omitting
  The set value in the default system configuration file (factory setting: 4) applied.

19.14 Fixed-sized Memory Pool Information (memorypool[])

Here, each fixed-sized memory pool is defined.

Format

Parentheses < >show the user input part.

```
memorypool[ <1. ID number> ] {
    name       = <2. ID name (name)>;
    siz_block  = <3. The size of the fixed-sized memory block (siz_block)>;
    num_block  = <4. The number of the fixed-sized memory block (num_block)>;
    section    = <5. Section name assigned to the memory pool area (section)>
    wait_queue = <6. Wait queue attribute (wait_queue)>;
};
```

1) ID number

- Description
  Define the fixed-sized memory pool ID number.
- Definition format
  Numeric value
- Definition range
  From 1 to 255
- When omitting
  The cfg600 assigns the ID number automatically.
- Note
  The ID numbers must be assigned without an omission beginning with 1. Therefore, when specifying an ID number, be sure that the specified value is equal to or less than the number of objects defined.

2) ID name (*name*)

- Description
  Define the ID name. The specified ID name is output to the system information header file (kernel_id.h) in the form of the following.

```
#define    <ID name>    <ID number>
```

- Definition format
  Symbol
- Definition range
  -
- When omitting
  Cannot be omitted.

3) The size of the fixed-sized memory block (*siz_block*)

- Description
  Define the size of the fixed-sized memory block in bytes.
- Definition format
  Numeric value
- Definition range
  From 1 to 65535
- When omitting
  The set value in the default system configuration file (factory setting: 256) applied.

4) The number of the fixed-sized memory block (*num_block*)

- Description
  Define the number of the fixed-sized memory block.
- Definition format
  Numeric value
- Definition range
  From 1 to 65535
- When omitting
  The set value in the default system configuration file (factory setting: 1) applied.

5) Section name assigned to the memory pool area (*section*)

- Description
  Define the section name to be assigned to the fixed-sized memory pool area.
  The cfg600 generates the fixed-sized memory pool area with the size calculated by “*siz_block * num_block*” to the section specified by *section*. The section attribute is “DATA”, and the alignment number is 4.
  When linking, be sure to locate this section in the RAM area. Note, this section must not be located to address 0.
- Definition format
  Symbol
- Definition range
  -
- When omitting
  The set value in the default system configuration file (factory setting: “BRI_HEAP”) applied.

6) Wait queue attribute (*wait_queue*)

- Description
  Define the wait queue attribute.
- Definition format
  Symbol
- Definition range
  Select either of the following:

  TA_TFIFO: FIFO order

  TA_TPRI: Task priority order
  Among tasks with the same priority, they are queued in FIFO order.
- When omitting
  The set value in the default system configuration file (factory setting: “TA_TFIFO”) applied.

19.15 Variable-sized Memory Pool Information (variable_memorypool[])

Here, each variable-sized memory pool is defined.

Format

Parentheses < >show the user input part.

```
variable_memorypool[ <1. ID number> ] {
    name        = <2. ID name (name)>;
    heap_size   = <3. The size of the variable-sized memory pool (heap_size)>;
    num_block   = <4. Upper limit of the variable-sized memory block (max_memsize)>;
    section     = <5. Section name assigned to the memory pool area (mpl_section)>
};
```

1) ID number

- Description
  Define the variable-sized memory pool ID number.
- Definition format
  Numeric value
- Definition range
  From 1 to 255
- When omitting
  The cfg600 assigns the ID number automatically.
- Note
  The ID numbers must be assigned without an omission beginning with 1. Therefore, when specifying an ID number, be sure that the specified value is equal to or less than the number of objects defined.

2) ID name (*name*)

- Description
  Define the ID name. The specified ID name is output to the system information header file (kernel_id.h) in the form of the following.

```
#define    <ID name>    <ID number>
```

- Definition format
  Symbol
- Definition range
  -
- When omitting
  Cannot be omitted.

3) The size of the variable-sized memory pool (*heap_size*)

- Description
  Define the size of the variable-sized memory pool area in bytes.
- Definition format
  Numeric value
- Definition range
  From 24 to 0x10000000
- When omitting
  The set value in the default system configuration file (factory setting: 1024) applied.

4) Upper limit of the variable-sized memory block (*max_memsize*)

- Description
  Define the upper limit of an acquirable memory block size in bytes.
- Definition format
  Numeric value
- Definition range
  From 1 to 0xBFFFFF4
- When omitting
  The set value in the default system configuration file (factory setting: 36) applied.
- Note
  Refer to "7.3.2 Size of Variable-sized memory block." for the size of the variable-sized memory blocks.

5) Section name assigned to the memory pool area (*mpl_section*)

- Description
  Define the section name to be assigned to the variable-sized memory pool area.
  The cfg600 generates the variable-sized memory pool area with the size specified by *heap_size* to the section specified by *mpl_section*. The section attribute is "DATA", and the alignment number is 4.
  When linking, be sure to locate this section in the RAM area. Note, this section must not be located to address 0.
- Definition format
  Symbol
- Definition range
  -
- When omitting
  The set value in the default system configuration file (factory setting: "BRI_HEAP") applied.

19.16 Cyclic Handler Information (cyclic_hand[])

Here, each cyclic handler is defined.

Format

Parentheses < >show the user input part.

```
cyclic_hand[ <1. ID number> ] {
    name             = <2. ID name (name)>;
    entry_address    = <3. Cyclic handler entry address (entry_address)>;
    interval_counter = <4. Activation cycle (interval_counter)>;
    start            = <5. Initial state (start)>;
    phs_counter      = <6. Activation phase (phs_counter)>;
    phsatr           = <7. TA_PHS attribute (phsatr)>;
    exinf            = <8. Extended information (exinf)>;
};
```

1) ID number

- Description
  Define the cyclic handler ID number.

- Definition format
  Numeric value

- Definition range
  From 1 to 255

- When omitting
  The cfg600 assigns the ID number automatically.

- Note
  The ID numbers must be assigned without an omission beginning with 1. Therefore, when specifying an ID number, be sure that the specified value is equal to or less than the number of objects defined.

2) ID name (*name*)

- Description
  Define the ID name. The specified ID name is output to the system information header file (kernel_id.h) in the form of the following.

```
        #define    <ID name>    <ID number>
```

- Definition format
  Symbol

- Definition range
  -

- When omitting
  Cannot be omitted.

3) Cyclic handler entry address (*entry_address*)

- Description
  Define the starting function of the cyclic handler.

- Definition format
  Symbol

- Definition range
  -

- When omitting
  Cannot be omitted.

4) Activation cycle (*interval_counter*)

- Description
  Define the activation cycle in millisecond.
- Definition format
  Numeric value
- Definition range
  From 1 to (0x7FFFFFFF - system.tic_nume) / system.tic_deno
- When omitting
  The set value in the default system configuration file (factory setting: 1) applied.

5) Initial state (*start*)

- Description
  Define the initial state of the cyclic handler.
- Definition format
  Symbol
- Definition range
  Select either of the following:

  OFF: Non operational stat (The TA_STA attribute is not specified.)

  ON: Operational state (The TA_STA attribute is specified.)
- When omitting
  The set value in the default system configuration file (factory setting: “OFF”) applied.

6) Activation phase (*phs_counter*)

- Description
  Define the activation phase in millisecond
- Definition format
  Numeric value
- Definition range
  From 0 to *interval_counter*
- When omitting
  The set value in the default system configuration file (factory setting: 0) applied.

7) TA_PHS attribute (*phsatr*)

- Description
  Define the attribute concerning the activation phase.
- Definition format
  Symbol
- Definition range
  Select either of the following:

  OFF: Not preserve the activation phase. (The TA_PHS attribute is not specified.)

  ON: Preserve the activation phase. (The TA_PHS attribute is specified.)
- When omitting
  The set value in the default system configuration file (factory setting: “OFF”) applied.

8) Extended information (*exinf*)

- Description
  Define the extended information of the cyclic handler.

- Definition format
  Numeric value
- Definition range
  From 0 to 0xFFFFFFFF
- When omitting
  The set value in the default system configuration file (factory setting: 0) applied.
- Note
  The extended information is passed to the cyclic handler.

19.17 Alarm Handler Information (alarm_handl[])

Here, each alarm handler is defined.

Format

Parentheses < >show the user input part.

```
alarm_hand[ <1. ID number> ] {
    name            = <2. ID name (name)>;
    entry_address   = <3. Alarm handler entry address (entry_address)>;
    exinf           = <4. Extended information (exinf)>;
};
```

1) ID number

- Description
  Define the alarm handler ID number.

- Definition format
  Numeric value

- Definition range
  From 1 to 255

- When omitting
  The cfg600 assigns the ID number automatically.

- Note
  The ID numbers must be assigned without an omission beginning with 1. Therefore, when specifying an ID number, be sure that the specified value is equal to or less than the number of objects defined.

2) ID name (*name*)

- Description
  Define the ID name. The specified ID name is output to the system information header file (kernel_id.h) in the form of the following.

  ```
  #define   <ID name>   <ID number>
  ```

- Definition format
  Symbol

- Definition range
  \-

- When omitting
  Cannot be omitted.

3) Alarm handler entry address (*entry_address*)

- Description
  Define the starting function of the alarm handler.

- Definition format
  Symbol

- Definition range
  \-

- When omitting
  Cannot be omitted.

4) Extended information (*exinf*)

- Description
  Define the extended information of the alarm handler.
- Definition format
  Numeric value
- Definition range
  From 0 to 0xFFFFFFFF
- When omitting
  The set value in the default system configuration file (factory setting: 0) applied.
- Note
  The extended information is passed to the alarm handler.

19.18 Relocatable Vector Information (interrupt_vector[])

Here, each interrupt handler for relocatable vector of the RX MCU is defined.
If any interrupt occurs whose vector number is not defined here, the system goes down.
Note, the cfg600 does not generate code to initialize the interrupt control registers, the causes of interrupts, etc. for the interrupts defined here. These initialization need to be implemented in the application.

Note Since the vector number from 1 to 8 are reserved by the RI600V4, do not define these vectors. And do not define the vectors which are reserved by the MCU specification.

Format

Parentheses < >show the user input part.

```
interrupt_vector[ <1. Vector number> ] {
    entry_address = <2. Interrupt handler entry address (entry_addreess)>;
    os_int        = <3. Kernel interrupt specification (os_int)>;
    pragma_switch = <4. Switch passed to pragma directive (pragma_switch)>;
};
```

1) Vector number

- Description
  Define the vector number.
- Definition format
  Numeric value
- Definition range
  From 0 to 255
- When omitting
  Cannot be omitted.

2) Interrupt handler entry address (*entry_addreess*)

- Description
  Define the starting function of the interrupt handler.
- Definition format
  Symbol
- Definition range
  -
- When omitting
  Cannot be omitted.

3) Kernel interrupt specification (*os_int*)

- Description
  Interrupts whose interrupt priority level is lower than or equal to the Kernel interrupt mask level (system_IPL) must be defined as the kernel interrupt, and the other interrupts must be defined as the non-kernel interrupt.
  Note, when the Kernel interrupt mask level (system_IPL) is 15, all interrupts for relocatable vector must be defined as the kernel interrupt.
- Definition format
  Symbol
- Definition range
  Select either of the following:

  YES: Kernel interrupt

  NO: Non-kernel interrupt

- When omitting
  Cannot be omitted.

4) Switch passed to pragma directive (*pragma_switch*)

- Description
  The cfg600 outputs “#pragma interrupt” directive to handle the function specified by *entry_address* as a interrupt function to the system information header file kernel_id.h.
  The switches passed to this pragma directive should be specified for *pragma_switch*.

- Definition format
  Symbol

- Definition range
  The following can be specified. To specify multiple choices, separate each with a comma. However, “ACC” and “NOACC” cannot be specified at the same time.

  E: The “enable” switch that permits a multiple interrupt is passed.

  F: The “fint” switch that specifies a fast interrupt is passed. Note, a fast interrupt must be handled as non-kernel interrupt (os_int = NO).

  S: The “save” switch that limits the number of registers used in the interrupt handler is passed.

  ACC: The “acc” switch that guarantees the ACC register in the interrupt handler is passed.

  NOACC: The “no_acc” switch that does not guarantee the ACC register in the interrupt handler is passed

- When omitting
  No switches are passed.

Note 1 Refer to Table 19-9 for the guarantee of the ACC register.

Table 19-9 Guarantee of the ACC Register

Setting of pragma_switch	“-save_acc” compiler option	
	Not specified	Specified
Neither “ACC” nor “NOACC” is not specified.	Neither “acc” nor “no_acc” switch is not passed. The ACC register is not guaranteed.	Neither “acc” nor “no_acc” switch is not passed. The ACC register is guaranteed.
“ACC” is specified.	The “acc” switch is passed. The ACC register is guaranteed.	
“NOACC” is specified.	The “no_acc” switch is passed. The ACC register is not guaranteed.	

Note 2 When either “CMT0”, “CMT1”, “CMT2” or “CMT3” is defined as Selection of timer channel for base clock (timer), it is treated that “interrupt_vector[]” is implicitly defined by the following specification.

- Vector number
  - CMT0 : 28
  - CMT1 : 29
  - CMT2 : 30
  - CMT3 : 31
- *entry_address* : The entry address of the base clock interrupt processing routine in the RI600V4
- *os_int* : YES

- *pragma_switch* : E,ACC

19.19 Fixed Vector/Exception Vector Information (interrupt_fvector[])

Here, fixed vector table of the RXv1 architecture (address from 0xFFFFFF80 to 0xFFFFFFFF) / exception vector table of RXv2 architecture is defined.
Not only interrupt handler address but also the endian select register, etc., are included in fixed vector table/exception vector table.
All interrupt in fixed vector/exception vector is non-kernel interrupt.
In the RI600V4, the vector number is allocated according to the vector address as shown in Table 19-10. Table 19-10 also shows the setting of the vector to which the definition is omitted.
Note, the content of fixed vector table/exception vector table is different in each MCU. For details, refer to the hardware manual of the MCU used.
Note, the cfg600 does not generate code to initialize the interrupt control registers, the causes of interrupts, etc. for the interrupts defined here. These initialization need to be implemented in the application.

Table 19-10 Fixed Vector Table/Exception Vector table

<table>
<tr><th>Vector address [a]</th><th>Vector number</th><th>Example of factor
(different in each MCU)</th><th>When omitting</th></tr>
<tr><td>0xFFFFFF80</td><td>0</td><td>Endian select register</td><td>The following are set according to "-endian" compiler option.
- "-endian=little"
0xFFFFFFFF
- "-endian=big"
0xFFFFFFF8</td></tr>
<tr><td>0xFFFFFF84</td><td>1</td><td>(Reserved area)</td><td rowspan="15">0xFFFFFFFF</td></tr>
<tr><td>0xFFFFFF88</td><td>2</td><td>Option function select register 1</td></tr>
<tr><td>0xFFFFFF8C</td><td>3</td><td>Option function select register 0</td></tr>
<tr><td>0xFFFFFF90</td><td>4</td><td>(Reserved area)</td></tr>
<tr><td>0xFFFFFF94</td><td>5</td><td>(Reserved area)</td></tr>
<tr><td>0xFFFFFF98</td><td>6</td><td>(Reserved area)</td></tr>
<tr><td>0xFFFFFF9C</td><td>7</td><td>ROM code protection (flash memory)</td></tr>
<tr><td>0xFFFFFFA0</td><td>8</td><td rowspan="4">ID code protection on connection of the on-chip debugger (flash memory)</td></tr>
<tr><td>0xFFFFFFA4</td><td>9</td></tr>
<tr><td>0xFFFFFFA8</td><td>10</td></tr>
<tr><td>0xFFFFFFAC</td><td>11</td></tr>
<tr><td>0xFFFFFFB0</td><td>12</td><td>(Reserved area)</td></tr>
<tr><td>0xFFFFFFB4</td><td>13</td><td>(Reserved area)</td></tr>
<tr><td>0xFFFFFFB8</td><td>14</td><td>(Reserved area)</td></tr>
<tr><td>0xFFFFFFBC</td><td>15</td><td>(Reserved area)</td></tr>
</table>

Vector address [a]	Vector number	Example of factor (different in each MCU)	When omitting
0xFFFFFFC0	16	(Reserved area)	System down
0xFFFFFFC4	17	(Reserved area)	
0xFFFFFFC8	18	(Reserved area)	
0xFFFFFFCC	19	(Reserved area)	
0xFFFFFFD0	20	Privileged instruction exception	
0xFFFFFFD4	21	Access exception	
0xFFFFFFD8	22	(Reserved area)	
0xFFFFFFDC	23	Undefined instruction exception	
0xFFFFFFE0	24	(Reserved area)	
0xFFFFFFE4	25	Floating-point exception	
0xFFFFFFE8	26	(Reserved area)	
0xFFFFFFEC	27	(Reserved area)	
0xFFFFFFF0	28	(Reserved area)	
0xFFFFFFF4	29	(Reserved area)	
0xFFFFFFF8	30	Non-maskable interrupt	
0xFFFFFFFC	31	Reset	PowerON_Reset_PC()

a. The vector address in Table 19-10 is the address of fixed vector table in RXv1 architecture.
The address of exception vector table in RXv2 architecture is decided by EXTB register. The initial value of EXTB register at the time of reset is same as fixed vector table in RXv1 architecture. Refer to "FIX_INTERRUPT_VECTOR section" in section 2.6.4.

Format

Parentheses < >show the user input part.

```
interrupt_fvector[ <1. Vector number> ] {
    entry_address = <2. Interrupt handler entry address (entry_addreess)>;
    pragma_switch = <3. Switch passed to pragma directive (pragma_switch)>;
};
```

1) Vector number

- Description
  Define the vector number.

- Definition format
  Numeric value

- Definition range
  From 0 to 31

- When omitting
  Cannot be omitted.

2) Interrupt handler entry address (*entry_addreess*)

- Description
  Define the starting function of the interrupt handler or the set value to fixed vector/exception vector.
- Definition format
  Symbol or numeric value
- Definition range
  From 0 to 0xFFFFFFFF when a numeric value is specified.
- When omitting
  Cannot be omitted.

3) Switch passed to pragma directive (*pragma_switch*)

- Description
  The cfg600 outputs “#pragma interrupt” directive to handle the function specified by *entry_address* as a interrupt function to the system information header file kernel_id.h.
  The switches passed to this pragma directive should be specified for *pragma_switch*.
- Definition format
  Symbol
- Definition range
  The following can be specified. To specify multiple choices, separate each with a comma. However, “ACC” and “NOACC” cannot be specified at the same time.

S:	The “save” switch that limits the number of registers used in the interrupt handler is passed.
ACC:	The “acc” switch that guarantees the ACC register in the interrupt handler is passed.
NOACC:	The “no_acc” switch that does not guarantee the ACC register in the interrupt handler is passed

- When omitting
  No switches are passed.
- Note
  Refer to Table 19-9 for the guarantee of the ACC register.

19.20 RAM Capacity Estimation

Memory areas used and managed by the RI600V4 are broadly classified into six types of sections. Subsequent paragraphs explain BRI_RAM, BURI_HEAP, SURI_STACK and SI section.

- BRI_RAM section: The RI600V4's management data and data queue area.
- BRI_HEAP section: Default section for message buffer area, fixed-sized memory pool area and variable-sized memory pool area.
- SURI_STACK section: Default section for user stack area
- SI section: System stack area
- RRI_RAM section: The RI600V4's management data. The size is 4 bytes.
- BRI_TRCBUF section: This section is generated only when "Taking in trace chart by software trace mode" and "Kernel buffer" are selected in [Task Analyzer] tab. The size is specified in [Task Analyzer] tab.

19.20.1 BRI_RAM section

The RI600V4's management data is located in the BRI_RAM section.
The Table 19-11 shows the size calculation method for the BRI_RAM section (unit: bytes). In addition, actual size may become larger than the value computed by Table 19-11 for boundary adjustment.

Table 19-11 BRI_RAM Section Size Calculation Method

Object Name	Size Calculation Method (in bytes)
System control block	36 + 4 * down(*TMAX_TPRI* - 1) / 32 + 1) + *TMAX_TPRI* + *VTMAX_SEM* + 2 * *VTMAX_DTQ* + *VTMAX_FLG* + *VTMAX_MBX* + *VTMAX_MTX* + 2 * *VTMAX_MBF* + *VTMAX_MPF* + *VTMAX_MPL*
Task control block	24 * *VTMAX_TSK*
Semaphore control block	4 * *VTMAX_SEM* + down (*VTMAX_SEM* / 8 + 1) However, when *VTMAX_SEM* is 0, the size of the semaphore control block is 0.
Eventflag control block	8 * *VTMAX_FLG* + 2 * down (*VTMAX_FLG* / 8 + 1) However, when *VTMAX_FLG* is 0, the size of the eventflag control block is 0.
Data queue control block	6 * *VTMAX_DTQ* + down (*VTMAX_DTQ* / 8 + 1) + *DTQ_ALLSIZE* However, when *VTMAX_DTQ* is 0, the size of the data queue control block is 0.
Mailbox control block	8 * *VTMAX_MBX* + 2 * down (*VTMAX_MBX* / 8 + 1) However, when *VTMAX_MBX* is 0, the size of the mailbox control block is 0.
Mutex control block	*VTMAX_MTX* + down (*VTMAX_MTX* / 8 + 1) However, when *VTMAX_MTX* is 0, the size of the mutex control block is 0.
Message buffer control block	16 * *VTMAX_MBF*
Fixed-sized memory pool control block	8 * *VTMAX_MPF* + 2 * down (*VTMAX_MPF* / 8 + 1) + Σ (down(memorypool[].num_block / 8 + 1)) However, when *VTMAX_MPF* is 0, the size of the fixed-sized memory pool control block is 0.
Variable-sized memory pool control block	208 * *VTMAX_MPL*
Cyclic handler control block	8 * *VTMAX_CYH*
Alarm handler control block	8 * *VTMAX_ALH*
"Taking in trace chart by hardware trace mode" is selected in [Task Analyzer] tab	4
"Taking in trace chart by software trace mode" is selected in [Task Analyzer] tab	28
"Taking in long-statistics by software trace mode" is selected in [Task Analyzer] tab	1592 + 8 × (*VTMAX_TSK* + 1)

Note Each keyword in the size calculation methods has the following meaning.

TMAX_TPRI: The set value of Maximum task priority (priority) in System Information (system). The cfg600 outputs the macro of this name to the system information header file kernel_id.h.

VTMAX_TSK: The number of Task Information (task[]). The cfg600 outputs the macro of this name to the system information header file kernel_id.h.

VTMAX_SEM: The number of Semaphore Information (semaphore[]). The cfg600 outputs the macro of this name to the system information header file kernel_id.h.

VTMAX_FLG: The number of Eventflag Information (flag[]). The cfg600 outputs the macro of this name to the system information header file kernel_id.h.

VTMAX_DTQ: The number of Data Queue Information (dataqueue[]). The cfg600 outputs the macro of this name to the system information header file kernel_id.h.

DTQ_ALLSIZE: Total of size of data queue area. Concretely, it is calculated by the following expressions.
Σ dataqueue[].buffer_size * 4
Note, *DTQ_ALLSIZE* is 4 when this calculation result is 0.

VTMAX_MBX: The number of Mailbox Information (mailbox[]). The cfg600 outputs the macro of this name to the system information header file kernel_id.h.

VTMAX_MTX: The number of Mutex Information (mutex[]). The cfg600 outputs the macro of this name to the system information header file kernel_id.h.

VTMAX_MBF: The number of Message Buffer Information (message_buffer[]). The cfg600 outputs the macro of this name to the system information header file kernel_id.h.

VTMAX_MPF: The number of Fixed-sized Memory Pool Information (memorypool[]). The cfg600 outputs the macro of this name to the system information header file kernel_id.h.

VTMAX_MPL: The number of Variable-sized Memory Pool Information (variable_memorypool[]). The cfg600 outputs the macro of this name to the system information header file kernel_id.h.

VTMAX_CYH: The number of Cyclic Handler Information (cyclic_hand[]). The cfg600 outputs the macro of this name to the system information header file kernel_id.h.

VTMAX_ALH: The number of Alarm Handler Information (alarm_handl[]). The cfg600 outputs the macro of this name to the system information header file kernel_id.h.

19.20.2 BRI_HEAP section

The message buffer area, fixed-sized memory pool area and variable-sized memory pool area are located in the BRI_HEAP section. Note, when a message buffer, fixed-sized memory pool and variable-sized memory pool are defined, the area can be located into the user-specific section.
The size of the BRI_HEAP section is calculated by the total of following.

- Total size of message buffer area
  This is calculated about the definition of Message Buffer Information (message_buffer[]) that omits to specify "mbf_section" by the following expressions.

  Σ message_buffer[].mbf_size

- Total size of fixed-sized memory pool area
  This is calculated about the definition of Fixed-sized Memory Pool Information (memorypool[]) that omits to specify "section" by the following expressions.

  Σ (memorypool[].siz_block * memorypool[].num_block)

- Total size of variable-sized memory pool area
  This is calculated about the definition of Variable-sized Memory Pool Information (variable_memorypool[]) that omits to specify "mpl_section" by the following expressions.

  Σ variable_memorypool[].heap_size

19.20.3 SURI_STACK section

The user stack area is located in the SURI_STACK section. Note, when a task is defined, the user stack area can be located into the user-specific section.
The size of the SURI_STACK section is calculated about the definition of Task Information (task[]) that omits to specify "stack_section" by the following expressions.

Σ task[].stack_size

Note For estimation of stack size, refer to "APPENDIX D STACK SIZE ESTIMATION".

19.20.4 SI section

The system stack area is located in the SI section.
The system stack size is the same as a set value for System stack size (stack_size) in System Information (system).

Note For estimation of stack size, refer to "APPENDIX D STACK SIZE ESTIMATION".

19.21 Description Examples

The following describes an example for coding the system configuration file.

```
// System Definition
system{
    stack_size  = 1024;
    priority    = 10;
    system_IPL  = 4;
    message_pri = 1;
    tic_deno    = 1;
    tic_nume    = 1;
    context     = FPSW,ACC;
};

//System Clock Definition
clock{
    template    = rx610.tpl;    // Please modify when you use other than RX610
    timer       = CMT0;         // Please modify for your H/W environment
    timer_clock = 25MHz;        // Please modify for your H/W environment
    IPL         = 3;            // Please modify for your H/W environment
};

//Task Definition
task[]{
    name            = ID_TASK1;
    entry_address   = task1();
    initial_start   = ON;
    stack_size      = 512;
    priority        = 1;
//  stack_section   = STK1;
    exinf           = 1;
};

task[]{
    name            = ID_TASK2;
    entry_address   = task2();
    initial_start   = ON;
    stack_size      = 512;
    priority        = 2;
//  stack_section   = STK2;
    exinf           = 2;
};

// Semaphore Definition
semaphore[]{
    name            = ID_SEM1;
    max_count       = 1;
    initial_count   = 1;
    wait_queue      = TA_TPRI;
};

// Cyclic Handler Definition
cyclic_hand[] {
    name            = ID_CYC1;
    entry_address   = cyh1();
    interval_counter = 100;
    start           = ON;
    phsatr          = OFF;
    phs_counter     = 100;
    exinf           = 1;
};
```

```
// Alarm Handler (dummy) Definition
alarm_hand[] {
    name             = ID_ALM1;
    entry_address    = alh1();
    exinf            = 1;
};

// Interrupt Handler for "Taking in trace chart by software trace mode"
//  Please remove the commnets when "Taking in trace chart by software trace mode"
//   is selected.

// interrupt_vector[29]{                // CMT CH1
//      os_int           = NO;
//      entry_address    = _RIUSR_trcSW_interrupt();      // in trcSW_cmt.src
// };

// Interrupt Handler for "Taking in long-statistics by software trace mode"
//  Please remove the commnets when "Taking in long-statistics by software trace
//  mode" is selected.
// interrupt_vector[29]{                // CMT CH1
//      os_int           = NO;
//      entry_address    = _RIUSR_trcLONG_interrupt();    // in trcLONG_cmt.src
// };

// Interrupt Handler (dummy) Definition
interrupt_vector[64]{
    os_int           = YES;
    entry_address    = inh64();
    pragma_switch    = E;
};
```

Note The RI600V4 provides sample source files for the system configuration file.

CHAPTER 20 CONFIGURATOR cfg600

This chapter explains configurator cfg600.

20.1 Outline

To build systems (load module) that use functions provided by the RI600V4, the information storing data to be provided for the RI600V4 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 RI600V4 provides a utility tool (configurator "cfg600") that converts a system configuration file which excels in descriptiveness and readability into information files.
The cfg600 reads the system configuration file as a input file, and then outputs information files.
The information files output from the cfg600 are explained below.

- System information header file (kernel_id.h)
  An information file that contains the correspondence between object names (task names, semaphore names, or the like) described in the system configuration file and IDs.

- Service call definition file (kernel_sysint.h)
  The declaration for issuing service calls by using INT instruction is described in this file. This file is included by kernel.h.

- ROM definition file (kernel_rom.h), RAM definition file (kernel_ram.h)
  These files contain the RI600V4 management data. These files must be included only by the boot processing source file. For details, refer to "16.2.1 Boot processing function (PowerON_Reset_PC( ))".

- System definition file (ri600.inc)
  The system definition file is included by the table file (ritable.src) which is generated by the mktitbl.

- Vector table template file (vector.tpl)
  The vector table template file is input to the mkritbl.

- CMT timer definition file (ri_cmt.h)
  When either of CMT0, CMT1, CMT or CMT3 is specified for Selection of timer channel for base clock (timer) for in Base Clock Interrupt Information (clock), the Template file (template) is retrieved from the folder indicated by the environment variable "LIB600", and the retrieved file is output after it is renamed to "ri_cmt.h". The CMT timer definition file must be included only by the boot processing source file. For details, refer to "16.2.1 Boot processing function (PowerON_Reset_PC( ))".

20.2 Start cfg600

20.2.1 Start cfg600 from command line

It is necessary to set the environment variable “LIB600” to “<ri_root>\lib600” beforehand.
The following is how to activate the cfg600 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 options enclosed in “[ ]” can be omitted.

```
C>  cfg600.exe Δ [-U]  Δ [-v] Δ [-V] Δ file <Enter>
```

The output files are generated to the current folder.

The details of each option are explained below:

- -U
  When an undefined interrupt occurs, the system down is caused. When -U option is specified, the vector number will be transferred to the system down routine (refer to “CHAPTER 13 SYSTEM DOWN”). This is useful for debugging. However, the kernel code size increases by about 1.5 KB.

- -v
  Show a description of the command option and details of its version.

- -V
  Show the creation status of files generated by the cfg600.

- *file*
  Specifies the system configuration file name to be input. If the filename extension is omitted, the extension “.cfg” is assumed.

Note <ri_root> indicates the installation folder of RI600V4.
The default folder is “C:\Program Files\Renesas Electronics\CubeSuite+\RI600V4”.

20.2.2 Start cfg600 from CubeSuite+

This is started when CubeSuite+ performs a build, in accordance with the setting on the Property panel, on the [System Configuration File Related Information] tab.

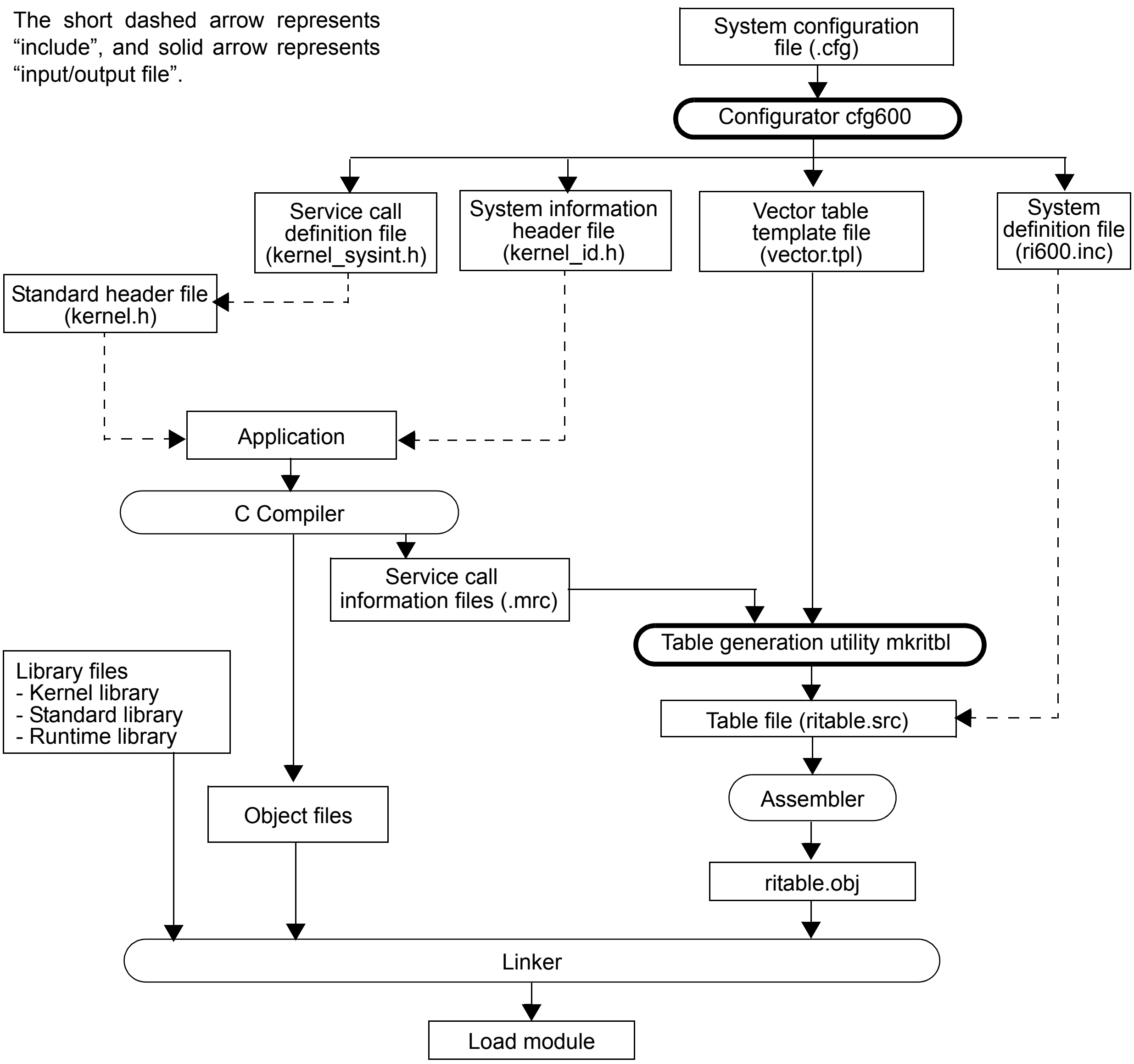
CHAPTER 21 TABLE GENARATION UTILITY mkritbl

This chapter explains the table generation utility mkritbl.

21.1 Outline

The utility mkritbl is a command line tool that after collecting service call information used in the application, generates service call tables and interrupt vector tables.
When compiling applications, the service call information files (.mrc) that contains the service call information to be used are generated. The mkribl reads the service call information files, and generates the service call table to be linked only the service calls used in the system.
Furthermore, the mkritbl generates an interrupt vector table based on the vector table template files generated by the cfg600 and the service call information files.

Figure 21-1 Outline of mkritbl



21.2 Start mkritbl

21.2.1 Start mkritbl from command line

It is necessary to set the environment variable "LIB600" to "<ri_root>\lib600" beforehand.
The following is how to activate the mkritbl 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 options enclosed in "[ ]" can be omitted.

```
C>   mkritbl.exe Δ [path] <Enter>
```

The output files are generated to the current folder.

The details of each option are explained below:

- *path*
  Specifies the service call information file or the path to the folder where the service call information files are retrieved. Note, when a folder path is specified, the sub folder is not retrieved.
  The mkritbl makes the current folder a retrieval path regardless of this specification.

Note <ri_root> indicates the installation folder of RI600V4.
The default folder is "C:\Program Files\Renesas Electronics\CubeSuite+\RI600V4".

21.2.2 Start mkritbl from CubeSuite+

This is started when CubeSuite+ performs a build, in accordance with the setting on the Property panel, on the [System Configuration File Related Information] tab.

21.3 Notes

Refer to "2.6.1 Service call information files and "-ri600_preinit_mrc" compiler option".

APPENDIX A WINDOW REFERENCE

This appendix explains the window/panels that are used when the activation option for the configurator cfg600 and the table generation utility mkritbl is specified from the integrated development environment CubeSuite+.

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 CubeSuite+ 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

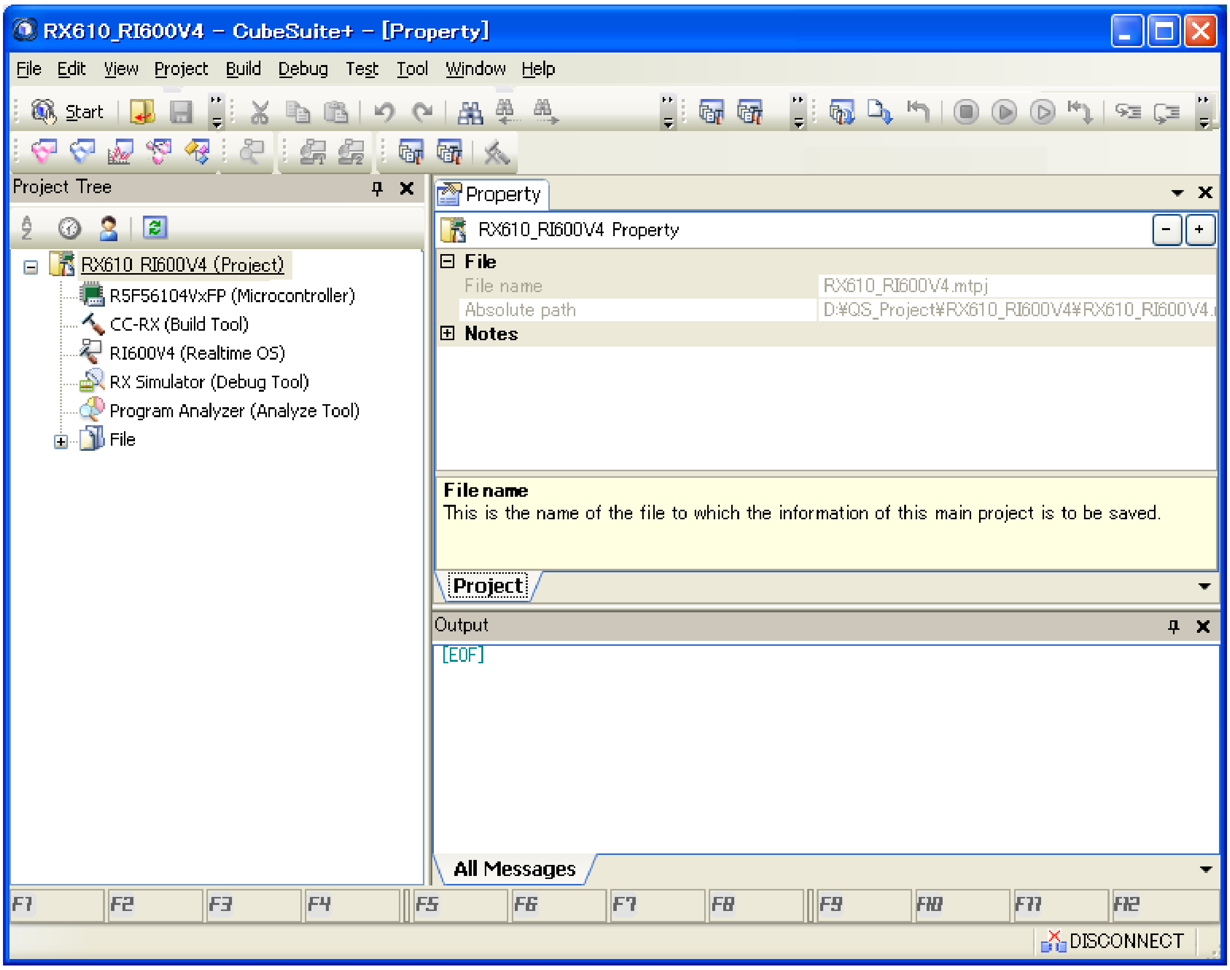
Outline

This is the first window to be open when CubeSuite+ 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 CubeSuite+] -> [CubeSuite+]

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. Note that this menu is disabled when the debug tool is not connected.
Task Analyzer 1	Opens the Realtime OS Task Analyzer 1 panel. Note that this menu is disabled when the debug tool is not connected.
Task Analyzer 2	Opens the Realtime OS Task Analyzer 2 panel 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. Note that this button is disabled when the debug tool is not connected.

- Realtime OS Task Analyzer toolbar

	Opens the Realtime OS Task Analyzer 1 panel Note that this button is disabled when the debug tool is not connected.
	Opens the Realtime OS Task Analyzer 2 panel 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 “CubeSuite+ Integrated Development Environment User's Manual: RX Build” for details about the Output panel.

Project Tree panel

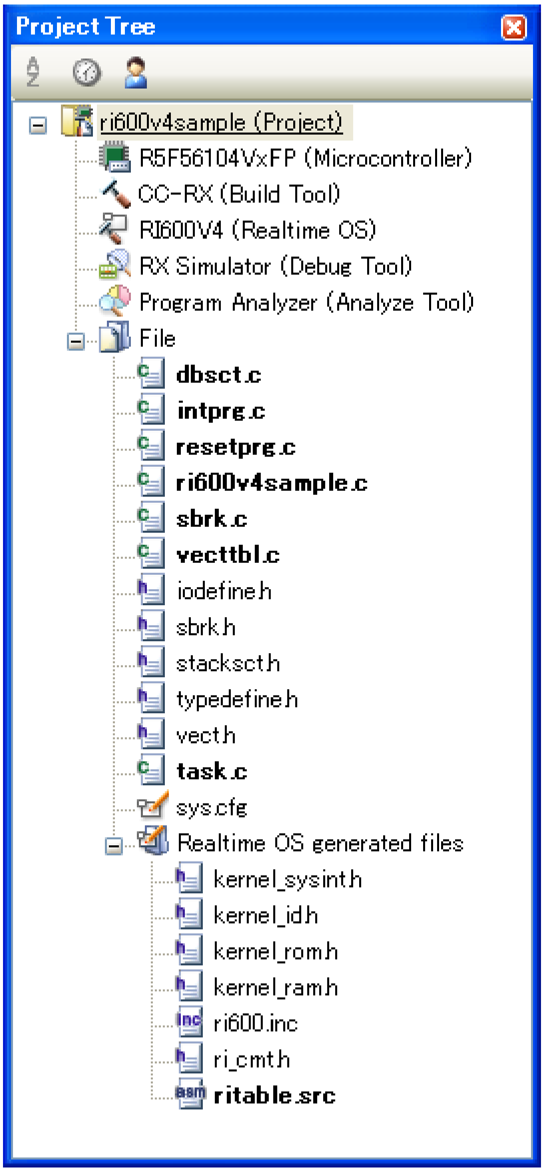
Outline

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
RI600V4 (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. - System information header file (kernel_id.h) - Service call definition file (kernel_sysint.h - ROM definition file (kernel_rom.h) - RAM definition file (kernel_ram.h) - System definition file (ri600.inc) - vector table template file (vector.tpl) - CMT timer definition file (ri_cmt.h) 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.

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. Note that this menu is only displayed when a system information table file or an entry file is selected. 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. Note that this menu is disabled when multiple files are selected.
Open with Internal Editor...	Opens the selected file with the Editor panel. 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. 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. 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. The file itself is not deleted from the file system. Note that this menu is disabled when the build tool is in operation.
Copy	Copies the selected file to the clipboard. 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. The actual file is also renamed.
Property	Displays the selected file's property on the Property panel.

Property panel

Outline

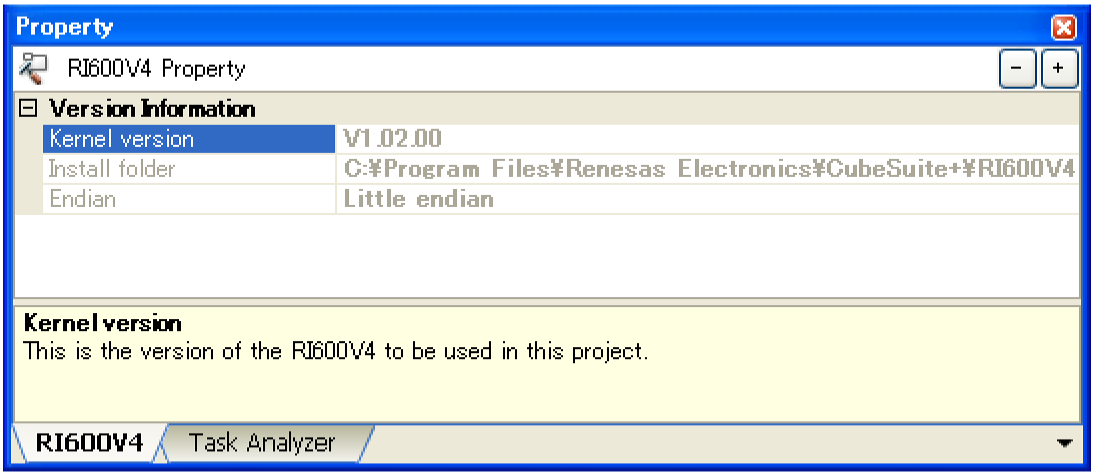
This panel is used to display the detailed information on the Realtime OS node, system configuration file, or the like that is selected on the Project Tree panel by every category and change the settings of the information.

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
  - [RI600V4] 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 or entry 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

Note1 See “CubeSuite+ Integrated Development Environment User's Manual: RX 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. 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. For the [Individual Assemble Options] tab, restores to the configuration of the general option.

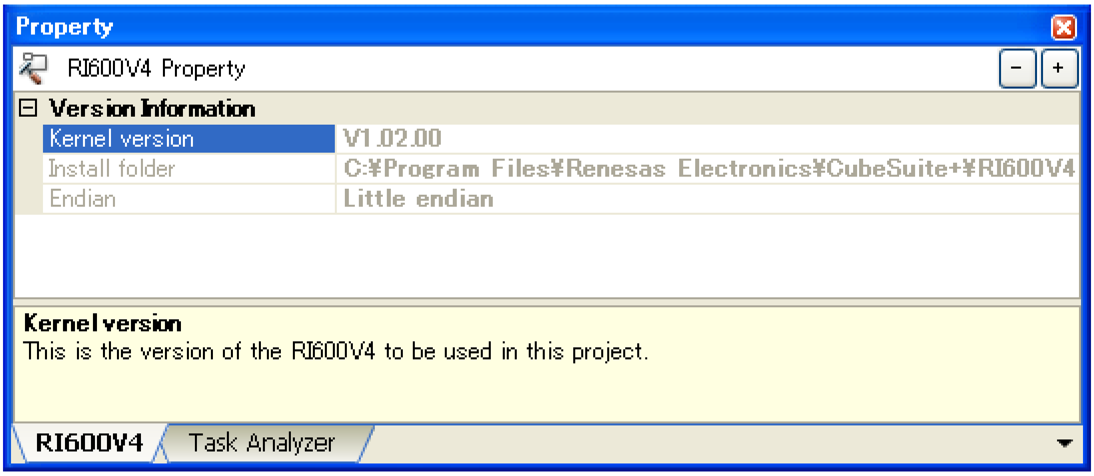
[RI600V4] tab

Outline

This tab shows the detailed information on RI600V4 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 RI600V4 are displayed.

Kernel version	Display the version of RI600V4 to be used.	
	Default	The version of the installed RI600V4
	How to change	Changes not allowed
Install folder	Display the folder in which RI600V4 to be used is installed with the absolute path.	
	Default	The folder in which RI600V4 to be used is installed
	How to change	Changes not allowed
Endian	Display the endian set in the project. Display the same value as the value of the [Select endian] property of the build tool.	
	Default	The endian in the property of the build tool
	How to change	Changes not allowed

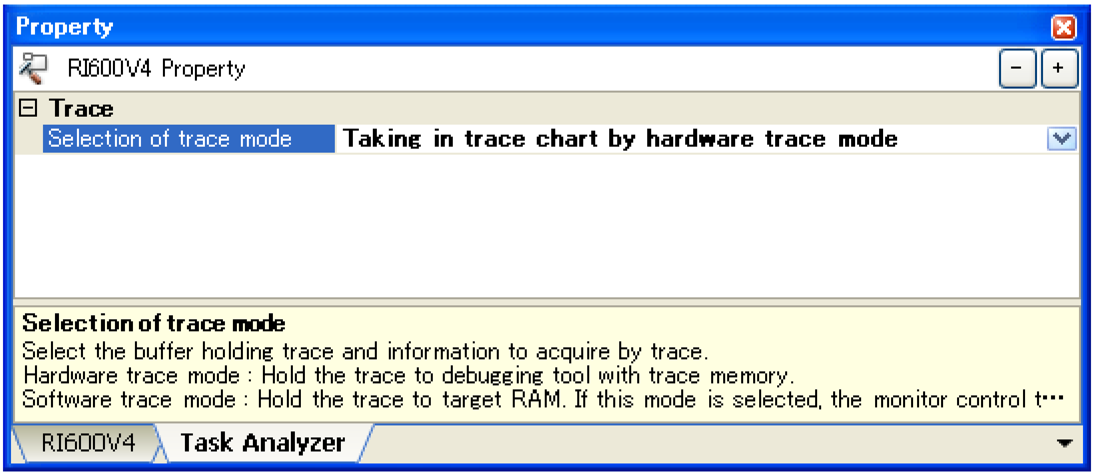
[Task Analyzer] tab

Outline

This tab sets up REALTIME OS TASK ANALYZER.

- Version Information

Display image



Explanation of each area

1) [Trace]

Sets up the trace mode of REALTIME OS TASK ANALYZER. According to this setup, the build-options shown in "2.6.6 Options for Realtime OS Task Analyzer" are set up automatically. Note, this automatic setting function is not being interlocked with corresponding property panel of a function. For this reason, don't change the contents set up automatically in corresponding property panel of a function.

Selection of trace mode	Select trace mode of Realtime OS Task Analyzer		
	Default	Taking in trace chart by hardware trace mode	
	How to change	Select from the drop-down list.	
	Restriction	Not tracing	Can not use Realtime OS Task Analyzer
		Taking in trace chart by hardware trace mode	The trace information is collected in the trace memory which emulator or simulator has.
		Taking in trace chart by software trace mode	The trace information is collected in the trace buffer secured on the user memory area. To use this mode, implementation of user-own coding module and setup of the system configuration file are required. For details, refer to chapter 15.3.1.
		Taking in long-statistics by software trace mode	The trace information is collected in the RI600V4's variable secured on the user memory area. To use this mode, implementation of user-own coding module and setup of the system configuration file are required. For details, refer to chapter 15.3.2.
Operation after used up the buffers	Select the operation after user up the trace buffer. This item is displayed only when "Taking in trace chart by software trace mode" is selected.		
	Default	Continue to exection while the buffers overwriting	
	How to change	Select from the drop-down list.	
	Restriction	Continue to exection while the buffers overwriting	It is overwritten sequentially from old information.
		Stop the trace taking in	The RI600V4 stops tracing.
Buffer size	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. This item is displayed only when "Taking in trace chart by software trace mode" is selected.		
	Default	0x100	
	How to change	Directly enter to the text box. Only a hexadecimal number can be entered.	
	Restriction	From 0x10 to 0x0FFFFFFF	
Select the buffer	Select the buffer. This item is displayed only when "Taking in trace chart by software trace mode" is selected.		
	Default	Kernel buffer	
	How to change	Select from the drop-down list.	
	Restriction	Kernel buffer	The trace buffer with the specified size is generated in the BRI_TRCBUF section when building.
		Another buffer	The buffer address is specified by the following clause.
Buffer address	Specify the start address of the "Another buffer" by immediate value. Area with "Buffer size" (bytes) from "Buffer address" is used as the trace buffer. Please be careful not to overlap with other program or data area. This item is displayed only when "Another buffer" is selected.		
	Default	0x0	
	How to change	Directly enter to the text box. Only a hexadecimal number can be entered	
	Restriction	From 0x0 to 0xFFFFFFFF, and "Buffer address" + "Buffer size" must not exceeds 0xFFFFFFFF.	

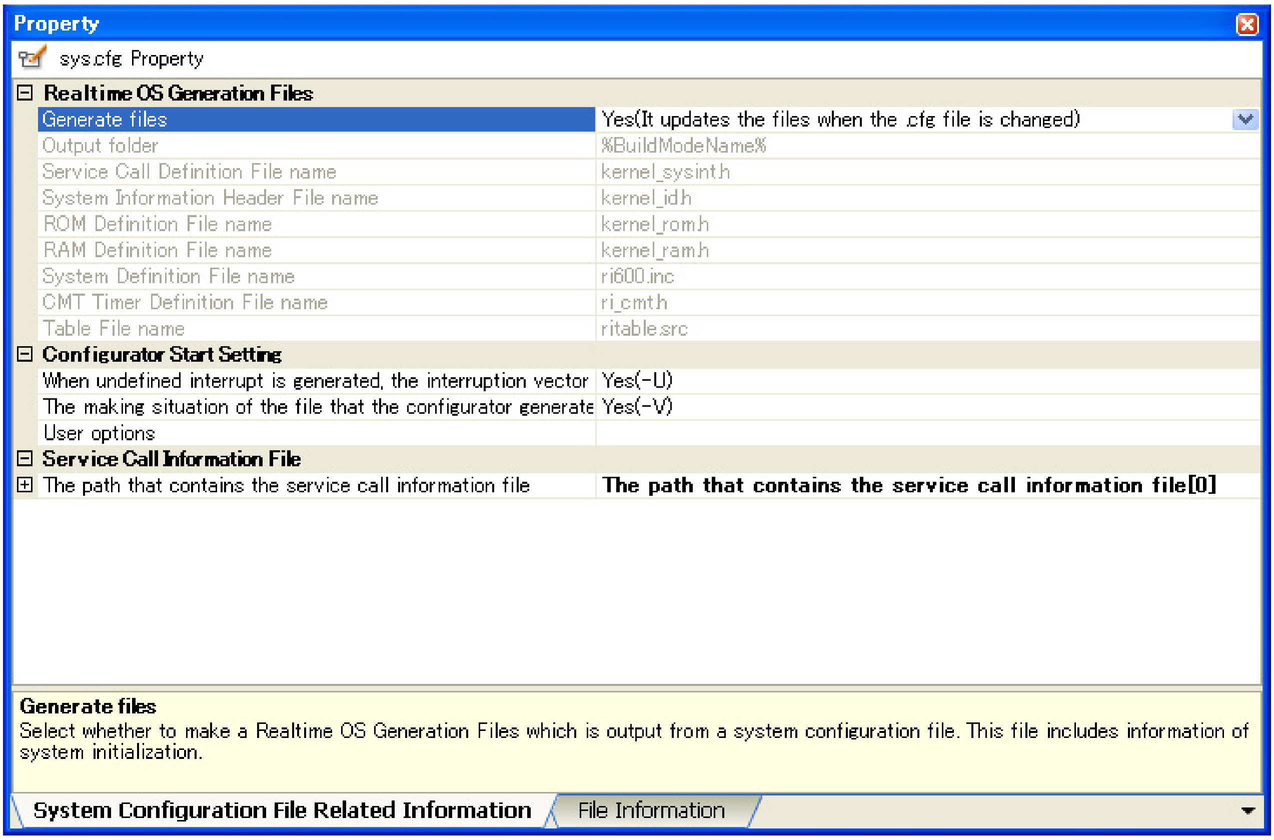
[System Configuration File Related Information] tab

Outline

This tab shows the detailed information on the using system configuration file categorized by the following and the configuration can be changed.

- Realtime OS Generation Files
- Configurator Start Setting
- Service Call Information File

Display image



Explanation of each area

1) [Realtime OS Generation Files]

The detailed information on the RI600V4 generation files are displayed and the configuration can be changed.

<table>
<tr><td rowspan="5">Generate files</td><td colspan="3">Select whether to generate realtime OS generation files and whether to update the realtime OS generation files 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)</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(It updates the file when the .cfg file is changed)</td><td>Generates new realtime OS generation files and displays them on the project tree.
If the system configuration file is changed when there are already realtime OS generation files, then realtime OS generation files are updated.</td></tr>
<tr><td>No(It does not register the file to the project)</td><td>Does not generate realtime OS generation files and does not display them on the project tree.
If this item is selected when there are already realtime OS generation files, then the files themselves are not deleted.</td></tr>
<tr><td rowspan="3">Output folder</td><td colspan="3">Display the folder for outputting realtime OS generation files.</td></tr>
<tr><td>Default</td><td colspan="2">%BuildModeName%</td></tr>
<tr><td>How to change</td><td colspan="2">Changes not allowed</td></tr>
<tr><td rowspan="3">Service Call Definition File Name</td><td colspan="3">Display the name of the service call definition file that the cfg600 outputs.</td></tr>
<tr><td>Default</td><td colspan="2">kernel_sysint.h</td></tr>
<tr><td>How to change</td><td colspan="2">Changes not allowed</td></tr>
<tr><td rowspan="3">System Information Header File Name</td><td colspan="3">Display the name of the system information header file that the cfg600 outputs.</td></tr>
<tr><td>Default</td><td colspan="2">kernel_id.h</td></tr>
<tr><td>How to change</td><td colspan="2">Changes not allowed</td></tr>
<tr><td rowspan="3">ROM Definition FIle Name</td><td colspan="3">Display the name of the ROM definition file that the cfg600 outputs.</td></tr>
<tr><td>Default</td><td colspan="2">kernel_rom.h</td></tr>
<tr><td>How to change</td><td colspan="2">Changes not allowed</td></tr>
<tr><td rowspan="3">RAM Definition FIle Name</td><td colspan="3">Display the name of the RAM definition file that the cfg600 outputs.</td></tr>
<tr><td>Default</td><td colspan="2">kernel_ram.h</td></tr>
<tr><td>How to change</td><td colspan="2">Changes not allowed</td></tr>
<tr><td rowspan="3">System Definition FIle Name</td><td colspan="3">Display the name of the system definition file that the cfg600 outputs.</td></tr>
<tr><td>Default</td><td colspan="2">ri600.inc</td></tr>
<tr><td>How to change</td><td colspan="2">Changes not allowed</td></tr>
<tr><td rowspan="3">CMT Timer Definition File Name</td><td colspan="3">Display the name of the CMT timer definition file which is generated by the cfg600.</td></tr>
<tr><td>Default</td><td colspan="2">ri_cmt.h</td></tr>
<tr><td>How to change</td><td colspan="2">Changes not allowed</td></tr>
</table>

<table>
<tr><td rowspan="3">Table File Name</td><td colspan="2">Display the name of the table file that the mkritbl outputs..</td></tr>
<tr><td>Default</td><td>ritable.src</td></tr>
<tr><td>How to change</td><td>Changes not allowed</td></tr>
</table>

2) [Configurator Start Setting]

The start option of the configurator cfg600 can be specified.

<table>
<tr><td rowspan="5">When undefined interrupt is generated, the interruption vector number is passed to system down routine.</td><td colspan="3">When an undefined interrupt occurs, the system down is caused. When -U option is specified, the vector number will be transferred to the system down routine (refer to “CAHPTER 13 SYSTEM DOWN”). This is useful for debugging. However, the kernel code size increases by about 1.5 KB.</td></tr>
<tr><td>Default</td><td colspan="2">Yes(-U)</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(-U)</td><td>When undefined interrupt is generated, the interruption vector number is passed to system down routine.</td></tr>
<tr><td>No</td><td>When undefined interrupt is generated, the interruption vector number is not passed to system down routine.</td></tr>
<tr><td rowspan="5">The making situation of the file that the configurator generates is displayed.</td><td colspan="3">Select whether to display the creation status of files generated by the cfg600.</td></tr>
<tr><td>Default</td><td colspan="2">Yes(-U)</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(-U)</td><td>Display the creation status of files generated by the cfg600.</td></tr>
<tr><td>No</td><td>Do not display the creation status of files generated by the cfg600.</td></tr>
<tr><td rowspan="4">User options.</td><td colspan="3">Input the command line option directly.</td></tr>
<tr><td>Default</td><td colspan="2">-</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">Up to 259 characters</td></tr>
</table>

3) [Service Call Information File]

Specify the path where the table generation utility mkritbl retrieves the service call information files.

<table>
<tr><td rowspan="3">The path that contains the service call information file.</td><td colspan="2">Specifies the service call information file (.mrc) or the path to the folder where the service call information files are retrieved.
Note, when a folder path is specified, the sub folder is not retrieved.
When relative path is specified, the project folder is the base folder.
When absolute path is specified, the specified path is converted into the relative path which is based from the project folder. However, if the drive of the specified path is different from the drive of the project folder, this conversion is not done.
Note, the project folder is passed to the mkritbl regardless this setting.
The following place holder can be specified.
%BuildModeName% : Convert to the build mode name.</td></tr>
<tr><td>How to change</td><td>Edit by the Path Edit dialog box which appears when clicking the [...] button.</td></tr>
<tr><td>Restriction</td><td>Up to 259 characters
Note, when extension is not specified or the specified extension is not ".mrc", the specified path is interpreted as folder.</td></tr>
</table>

Note 1 Refer to "2.6.1 Service call information files and "-ri600_preinit_mrc" compiler option" for the service call information file.

Note 2 When using the "optimization for accesses to external variables" compiler option, the CubeSuite+ generates the folder to store object files and service call information files for 1st build, and specifies this folder path for [Service Call Information File] tacit.

Note 3 The service call information files are generated to the same folder as object files at compilation. Please change this item appropriately when you do the operation to which the output folder of object files is changed.

APPENDIX B FLOATING-POINT OPERATION FUNCTION

It is only when the "-fpu" option is specified that the compiler outputs floating-point arithmetic instructions.
If the "-chkfpu" option is specified in the assembler, the floating-point arithmetic instructions written in a program are detected as warning.

B.1 When Using Floating-point Arithmetic Instructions in Tasks

Make settings that include "FPSW" for Task context register (context) in System Information (system). As a result, the FPSW register is managed independently in each task.
The initial FPSW of task is initialized by the value according to compiler options to be used. For details, refer to "3.2.4 Internal processing of task".

B.2 When Using Floating-point Arithmetic Instructions in Handlers

It is necessary that the handler explicitly guarantee the FPSW register.
The initial FPSW value of handlers is undefined.
To guarantee and initialize the FPSW register, write a program as follows.

```
#include <machine.h>    // To use the intrinsic function get_fpsw() and set_fpsw(),
                        // include machine.h.
#include "kernel.h"
#include "kernel_id.h"
void handler (void)
{
    unsigned long old_fpsw;    // Declare variable for saving the FPSW register
    old_fpsw = get_fpsw () ;    // Save the FPSW register
    set_fpsw (0x00000100) ;     // Initialize the FPSW register if necessary
    /* Floating-point arithmetic operation */
    set_fpsw (old_fpsw) ;       // Restore the FPSW register
}
```

APPENDIX C DSP FUNCTION

When a MCU which support the DSP function is used, it is necessary to note the treatment of the ACC register (accumulator). Concretely, please note it as follows when you use the following DSP instructions which update ACC register.

- RXv1/RXv2 architecture common instruction
  MACHI, MACLO, MULHI, MULLO, RACW, MVTACHI, MVTACLO

- RXv2 architecture instructions
  EMACA, EMSBA, EMULA, MACLH, MSBHI, MSBLH, MSBLO, MULLH, MVTACGU, RACL,RDACL, RDACW

In no case does the compiler generate these instructions.
Note also that if the “-chkdsp” option is specified in the assembler, the DSP function instructions written in a program are detected as warning.

C.1 When Using DSP Instructions in Tasks

Make settings that include “ACC” for Task context register (context) in System Information (system). As a result, the ACC register is managed independently in each task.

C.2 When Using DSP Instructions in Handlers

If the application contains any tasks or interrupt handlers that use the above-mentioned DSP instructions, it is necessary that all of the interrupt handlers guarantee the ACC register. There are the following two method.

1) Use “-save_acc” compiler option

2) Specify “ACC” for “pragma_switch” in all interrupt handler definition (Relocatable Vector Information (interrupt_vector[]) and Fixed Vector/Exception Vector Information (interrupt_fvector[])).

APPENDIX D STACK SIZE ESTIMATION

If a stack overflows, the behavior of the system becomes irregular. Therefore, a stack must not overflow referring to this chapter.

D.1 Types of Stack

There are two types of stacks: the user stack and system stack. The method for calculating stack sizes differs between the user stack and system stack.

- User stack
  The stack used by tasks is called “User stack”. When a task is created by Task Information (task[]), the size and the name of the section where the stack is allocated are specified.

- System stack
  The system stack is used by handlers and the kernel. The system has only one system stack. The size is specified by System stack size (stack_size) in System Information (system). The section name of the system stack is “SI”.

D.2 Call Walker

The CubeSuite+ package includes “Call Walker” which is a utility tool to calculate stack size.
The Call Walker can display stack size used by each function tree.

D.3 User Stack Size Estimation

The quantity consumed of user stack for each task is calculated by the following expressions.

Quantity consumed of user stack = *treesz* + *ctxsz*

- *treesz*
  Size consumed by function tree that makes the task entry function starting point. (the size displayed by Call Walker).

- *ctxsz*
  Size for task context registers. This size is different according to the setting of Task context register (context) in System Information (system). Refer to Table D-1.

Table D-1 Size of Task Context Register

Setting of system.context	Compiler option “-isa”	Size of Task Contest Register
NO	-	68
FPSW	-	72
ACC	“-isa=rxv2”	92
	“-isa=rxv1” or not specify “-isa”	76
FPSW,ACC	“-isa=rxv2”	96
	“-isa=rxv1” or not specify “-isa”	80
MIN	-	44
MIN,FPSW	-	48
MIN,ACC	“-isa=rxv2”	68
	“-isa=rxv1” or not specify “-isa”	52
MON,FPSW,ACC	“-isa=rxv2”	72
	“-isa=rxv1” or not specify “-isa”	56

Note Compiler option “-isa” is supported by the compiler CC-RX V2.01 or later.

D.4 System Stack Size Estimation

The system stack is most consumed when an interrupt occurs during service call processing followed by the occurrence of multiple interrupts. The quantity consumed of system stack is calculated by the following expressions.

$$\text{Quantity consumed of system stack} = svcsz + \sum_{k=1}^{15} inthdrsz_k + sysdwnsz$$

- *svcsz*
  The maximum size among the service calls to be used in the all processing program. The value *svcsz* depends on the RI600V4 version. For details, refer to release notes.

- *inthdrsz*
  Size consumed by function tree that makes the interrupt handler entry function starting point. (the size displayed by Call Walker).
  The "k" means interrupt priority level. If there are multiple interrupts in the same priority level, the $inthdrsz_k$ should select the maximum size among the handlers.
  The size used by the base clock interrupt handler (the interrupt priority level is specified by Base clock interrupt priority level (IPL) in Base Clock Interrupt Information (clock)) is the maximum value in the following Please refer to the release notes for *clocksz1*, *clocksz2* and *clocksz3*.
  Don't have to add the size used by the base clock interrupt handler when base clock timer is not used (clock.timer = NOTIMER).

  - *clocksz1* + *cycsz*

  - *clocksz2* + *almsz*

  - *clocksz3*

    - *cycsz*
      Size consumed by function tree that makes the cyclic handler entry function starting point. (the size displayed by Call Walker).
      If there are multiple cyclic handlers, the *cycsz* should select the maximum size among the handlers.

    - *almsz*
      Size consumed by function tree that makes the alarm handler entry function starting point. the size displayed by Call Walker).
      If there are multiple alarm handlers, the *cycsz* should select the maximum size among the handlers.

- *sysdwnsz*
  Size consumed by function tree that makes the system down routine entry function starting point. (the size displayed by Call Walker) + 40. When the system down routine has never been executed, *sysdwnsz* is assumed to be 0.

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Oct 01, 2011	-	First Edition issued
1.01	Apr 01, 2012	-	“Priority”, “current priority” and “base priority” have been improved so that they may be used properly clearly.
		15	“Section information file” has been added to section 2.4.
		22	Expression of section 2.6.2 has been improved.
		29	Expression of section 3.2.2 has been improved.
		32	The “Note 2” has been corrected as follows, - “7-bit width” -> “8-bit width” - “the maximum count value 127” -> “the maximum count value 255”
		73, 75, 76, 262, 264, 266	The following description has been added to “There is a data in the data queue.” When there is a task in the transmission wait queue, this service call stores the data sent by the task in the top of the transmission wait queue and moves it from the WAITING state (data transmission wait state) to the READY state.
		93	The following description has been added. And this service call changes the current priority of the task to the ceiling priority of the target mutex. However, this service call does not change the current priority when the task has locked other mutexes and the ceiling priority of the target mutex is lower than or equal to the ceiling priority of the locked mutexes.
		95, 98, 293, 297	The description for the case “The task at the top of the transmission wait queue was forcedly released.” has been added to the table for “Sending WAITING State for a Message Buffer Cancel Operation”.
		113, 313	The Note has been deleted.
		117, 120, 318, 323	The description for the case “The task at the top of the transmission wait queue was forcedly released.” has been added to the table for WAITING State for a Variable-sized Memory Block Cancel Operation.
		-	The composition of CHAPTER 8 has been improved.
		128	Expression of section 8.6.4 has been improved.
		139	“8.9 Initialize Base Clock Timer” has been added.
		145	Expression of section 9.5 has been improved.
		147	Expression of section 9.7 has been improved.
		148	Expression of section 9.8 has been improved.
		149	Expression of section 9.9 has been improved.
		162	The description of “IPL” in Table 13-1. has been detailed more.
		162	Explanation of “type == -1” has been corrected by “Error when a kernel interrupt handler ends” from “Error when a interrupt handler ends”.
		167	Expression of section 14.7 has been improved.
		-	The composition of CHAPTER 16 has been improved.

Rev.	Date	Page	Summary
		184	"16.4 Section Initialization Function (_INITSCT( ))" has been added.
		185	"16.5 Registers in Fixed Vector Table/Exception Vector table" has been added.
		186	Data type of INT, UINT, VP_INT, ER_UINT and FLGPTN in Table 17-1 have been corrected.by "singed long" or "unsigned long" from "signed int" or "unsigned int".
		188	Expression for TA_TPRI and TA_MPRI in Table 17-2 have been improved.
		224	The following description has been added for "E_CTX error". - The invoking task is specified in the dispatching disabled state.
		230	The following description has been added for "E_CTX error". - This service call was issued in the status "PSW.IPL > kernel interrupt mask level".
		283, 284, 287	The conditional expression of "Ceiling priority violation" of E_ILUSE error has been corrected.
		284, 287	The following description has been added to "Explanation". When the mutex is locked, this service call changes the current priority of the invoking task to the ceiling priority of the target mutex. However, this service call does not change the current priority when the invoking task has locked other mutexes and the ceiling priority of the target mutex is lower than or equal to the ceiling priority of the locked mutexes.
		288	The following description has been added to "Explanation". And this service call changes the current priority of the task to the ceiling priority of the target mutex. However, this service call does not change the current priority when the task has locked other mutexes and the ceiling priority of the target mutex is lower than or equal to the ceiling priority of the locked mutexes.
		290	Conditions of E_CTX error have been corrected.
		301	The following description has been deleted for "E_CTX error". - This service call was issued in the dispatching disabled state.
		330, 331	An order of members in SYSTIM structure has been corrected.
		337	The "E_PAR" error has been added to sta_alm and ista_alm.
		346	The "Note 5" has been added.
		363	The prid returned by ref_ver and iref_ver has been corrected by "0x0003" from "0x0004".
		384	The definition range of "*max_count*" has been corrected as follows. From 0 to 65535
		394	The following description has been deleted in "5) Maximum message size (max_msgsz)". The specified value is rounded up to the multiple of four.
		397	The following description for "The size of the variable-sized memory pool (heap_size)" in section 19.15 has been deleted.
		405	The note 2 has been added.
		411	The following description has been added in section 19.20.1. "In addition, actual size may become larger than the value computed by Table 19-11 for boundary adjustment."

Rev.	Date	Description	
		Page	Summary
		411	The following coefficients in Table 19-11 have been corrected. - Data queue control block The coefficient “8” of the head of the formula has been corrected by “6”. - Variable-sized memory pool control block The coefficient “36” of the head of the formula has been corrected by “208”.
		412	The following description has been added for *DTQ_ALLSIZE*. “Note, *DTQ_ALLSIZE* is 4 when this calculation result is 0.”
1.02	Sep 1, 2012 (RI600V4 V1.02.00)	15	The description about user-own coding module for the Realtime OS task Analyzer has been added.
		24	The DRI_ROM and RRI_RAM sections have been added to Table 2-3.
		26	“2.6.6 Options for Realtime OS Task Analyzer” has been added.
		168	“CHAPTER 15 REALTIME OS TASK ANALYZER” has been added.
		190	With revision to V1.02.00, the definition value of TKERNEL_PRVER has been changed into 0x0120.
		410	The RRI_RAM and BRI_TRCBUF sections have been added.
		411	- “28” of the beginning of the formula of the “System control block” has been changed into “36.” - The item corresponding to the Realtime OS Task Analyzer has been added.
		422	The description of menu and toolbar corresponding to the Realtime OS Task Analyzer have been added.
		430	“[Task Analyzer] tab” has been added.
1.03	May 15, 2013 (RI600V4 V1.02.02)	14	“Note 2” has been added to “2.3 Coding System Configuration File”
		22	Explanation of “2.6.2 Compiler option for the boot processing file” was detailed.
		25	”2.6.5 Initialized data section” has been added.
		31,436	The specification of FPSW register when task processing is started has been changed.
		168,431	A setup of the system configuration file has been added as a required matter when software trace mode is used.
		184	Expression of section “16.4 Section Initialization Function (_INITSCT( ))” has been improved.

Rev.	Date	Description	
		Page	Summary
1.04	Sep 20, 2013 (RI600V4 V1.03.00)	23	With support of RXv2 architecture, the composition of kernel libraries have been changed.
		25, 178, 185, 407, etc.	With support of RXv2 architecture, the explanation about FIX_INTERRUPT_VECTOR section and EXTB register have been added or changed. Moreover, “fixed vector” has been replaced by “fixed vector/exception vector”.
		179	With support of RXv2 architecture, the explanation about compiler option “-isa” and “-cpu” have been added.
		183, 357	The explanation about starting of RI600V4 has been improved.
		190	With revision to V1.03.00, the definition value of TKERNEL_PRVER has been changed into 0x0130.
		375	With support of RXv2 architecture, Table 19-2 has been changed.
		379	The explanation of Table 19-7 has been improved.
		382	With support of RXv2 architecture, Table 19-8 has been changed.
		437	The RXv2 instructions have been added to DSP instructions which update ACC register.
		437	The description “All interrupt handlers explicitly guarantee the ACC register” has been deleted.
		439	With support of RXv2 architecture, Table D-1 has been changed.

RI600V4 Real-Time Operating System
User's Manual: Coding

Publication Date: Rev.1.00 Oct 01, 2011
Rev.1.04 Sep 20, 2013

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.
2880 Scott Boulevard Santa Clara, CA 95050-2554, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
1101 Nicholson Road, Newmarket, Ontario L3Y 9C3, Canada
Tel: +1-905-898-5441, Fax: +1-905-898-3220

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-651-700, Fax: +44-1628-651-804

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-65030, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
7th Floor, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100083, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 204, 205, AZIA Center, No.1233 Lujiazui Ring Rd., Pudong District, Shanghai 200120, China
Tel: +86-21-5877-1818, Fax: +86-21-6887-7858 / -7898

Renesas Electronics Hong Kong Limited
Unit 1601-1613, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2886-9318, Fax: +852 2886-9022/9044

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei, 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 906, 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 Korea Co., Ltd.
11F., Samik Lavied' or Bldg., 720-2 Yeoksam-Dong, Kangnam-Ku, Seoul 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141

Colophon 1.3

RI600V4

Renesas Electronics Corporation

R20UT0711EJ0104