MF1169-02

EPSON

CMOS 32-BIT SINGLE CHIP MICROCOMPUTER **E0C33 Family**

ROS33 REALTIME OS MANUAL
(ver. 2.1)

ENERGY SAVING EPSON

SEIKO EPSON CORPORATION

Preface

Written for those who develop applications using the E0C33 Family of microcomputers, this manual describes the functions provided by the Realtime OS ROS33 for the E0C33 Family, and also gives precautions on programming for this OS.

ROS33 is a realtime OS designed to the µITRON 3.0 specifications. For information and literature relating to µITRON, see the ITRON Home Page on the Internet.

English) http://tron.um.u-tokyo.ac.jp/TRON/ITRON/home-e.html
Japanese) http://tron.um.u-tokyo.ac.jp/TRON/ITRON/home-j.html
(Note: This address is effective as of July 1998.)

An English version of the µITRON 3.0 specifications is provided on the ROS33 disk.

Table of Contents

1 ROS33 Package

ROS33 is a realtime OS for the E0C33 Family of single-chip microcomputers based on µITRON 3.0. Using ROS33 in your design enables you to quickly and efficiently develop embedded applications for printers, PDAs, and various types of control equipment.

1.1 Features

The main features of ROS33 are listed below.

- Based on µITRON 3.0. System calls up to Level S (standard) are supported.

Number of tasks:	1 to 255
Priority levels:	1 to 9
Number of event flags:	1 to 255
Number of semaphores:	1 to 255
Number of mailboxes:	1 to 255
Number of message buffers:	1 to 255
Scheduling method:	Priority basis
Semaphore:	Count type
Event flag:	Byte type (8 bits)
Mailboxes:	Passed via pointers

- Compact and high-speed kernel optimized for use in the E0C33 Family

  Kernel size*1:

  1.7K bytes...... Level R supported, no error check
  2.4K bytes...... Level R supported, standard
  2.7K bytes...... Level R supported, debug kernel
  2.6K bytes...... Level S supported, no error check
  3.6K bytes...... Level S supported, standard
  3.8K bytes...... Level S supported, debug kernel

  Dispatch time*2:

  7.8 µs .... 33 MHz, when using only the internal ROM and internal RAM
  14.3 µs .... 33 MHz, when using external ROM (2 wait states) and internal RAM
  12.9 µs .... 20 MHz, when using only the internal ROM and internal RAM
  23.6 µs .... 20 MHz, when using external ROM (2 wait states) and internal RAM

  Maximum interrupt disable time*2:

  4.3 µs .... 33 MHz, when using only the internal ROM and internal RAM
  9.0 µs .... 33 MHz, when using external ROM (2 wait states) and internal RAM
  7.2 µs .... 20 MHz, when using only the internal ROM and internal RAM
  14.8 µs .... 20 MHz, when using external ROM (2 wait states) and internal RAM

  *1 Number of tasks = 8, number of priority levels = 8, number of event flags = 8, number of semaphore = 8 and number of mailboxes = 8

  *2 These values were evaluated using the ICE33 when tasks of the same priority were switched over by a rot_rdq system call.

  These are standard values for a guide and will vary according to the user's system environment and the make condition. The net value should be evaluated on the actual system.

- Programs can be developed in C and assembly language.
- Provided for each function as a modularized library
  When linking, only necessary modules are selected. This enables you to minimize the size of the compiled application.
- Comes with source code for each functional module
  The number of resources can be customized to suit your system specification.
- Multiple tasks can share a common stack area (when not processed in parallel)
  You can minimize the amount of RAM used in your system by your application.

Note

- Make sure that accurate verification is necessary before product shipping. Seiko Epson does not assume any liability of ant kind arising out of any inaccuracies contained in this middleware.
- Dealership of this middleware belongs to Seiko Epson. Resale pretension can not be given.
- No part of this files in this package except sample programs may be duplicated, distributed, reproduced, or reverse engineered in any form or by ant means without the written permission of Seiko Epson.

1.2 ROS33 Package Components

The ROS33 package contains the following items. When opening your ROS33 package, check to see that all of these items are included.

(1) Tool disk (CD-ROM)	1
(2) E0C33 Family ROS33 Realtime OS Manual (this manual)	1 each in Japanese and English
(3) Warranty card	1 each in Japanese and English

1.3 Installing ROS33

ROS33 needs to be linked with the user program as it is implemented. Therefore, make sure all tools of the "E0C33 Family C Compiler Package" have been installed in your computer and are ready to run before installing ROS33 files in your computer. The basic system configuration is described below.

• Personal computer: IBM PC/AT or compatible
(Pentium 90 MHz or better; we recommend that you have more than 32 MB of memory)
• OS: Windows 95, Windows 98, Windows NT 4.0, or later (Japanese or English version)

All the ROS33 files are supplied on one CD-ROM. Execute the self-extract file "ros33vXX.exe" on the CD-ROM to install the files. ("XX" in the file name represents the version number, for example, "ros33v10.exe" is the file of ROS33 ver. 1.0.)
When "ros33vXX.exe" is started up by double-clicking the file icon, the following dialog box appears.

Enter a path/folder name in the text box and then click [Unzip]. The specified folder will be created and all the files will be copied in the folder.
When the specified folder already exists on the specified path, the folder will be overwritten without prompting if [Overwrite Files Without Prompting] is checked.

The directory and file configurations after copying the floppy disk contents are shown below.

(root)¥	(Default: C:¥E0C33¥ROS33¥)		
	itron302.txt		µITON 3.0 specification (English version, edited by TRON Association)
	readmeja.txt		Supplementary explanation (in Japanese)
	readme.txt		Supplementary explanation (in English)
	lib¥	..... ROS33 library	
		ros33.lib	ROS33 library
		ros33g.lib	ROS33 library with debug information
	include¥	..... Include files	
		itron.h	ITRON common header file
		ros33.h	ROS33 definition file
	src¥	..... Source files	
		debug.c	C source file for debug functions
		flag.c	C source file for event flag functions
		intmng.c	C source file for interrupt management functions
		mailbox.c	C source file for mailbox functions
		ros33.c	ROS33 main C source file
		ros33asm.s	Assembly source file for dispatch and ret_int functions
		semapho.c	C source file for semaphore functions
		timemng.c	C source file for time management functions
		tskmng.c	C source file for task management functions
		tsksync.c	C source file for task-dependent synchronization functions

	internal.h	ROS33 data type definition file
	msgbuf.c	C source file for message buffer function
build¥	..... ROS33 build files	
	ros33.mak	make file for ROS33.lib generation
demo¥		
	..... Demonstration program and related files	
sample¥		
	..... Sample programs and related files	

2 Programming

This chapter gives an outline of ROS33, and then shows how to create an application program and how to customize ROS33.

2.1 Outline of µITRON and ROS33

µITRON is a realtime, multitask OS which has been developed primarily by the ITRON Technical Committee of the TRON Association as part of the TRON Project. The purpose of developing this OS was to improve realtime processing capabilities and program productivity in embedded systems incorporating single-chip microcomputers.

ROS33 is a µITRON 3.0 (current version) specification compliant kernel for the E0C33 Family of microcomputers. ROS33 supports Level R (required) and Level S (standard).

∗ Regarding Levels R and S

µITRON is classified into several levels by system call functionality. Level R (required) is the essential function for µITRON 3.0 (current version) specification kernels, and includes the basic system calls necessary for realtime, multitask OSs. Level S (standard) includes standard system calls for realtime, multitask OSs. In addition to these, two other levels are available: Level E (extended), which includes additional and extended functions, and Level C (CPU dependent), which depends on the CPU and system implementation.

Figure 2.1.1 shows a conceptual diagram of a system configuration.

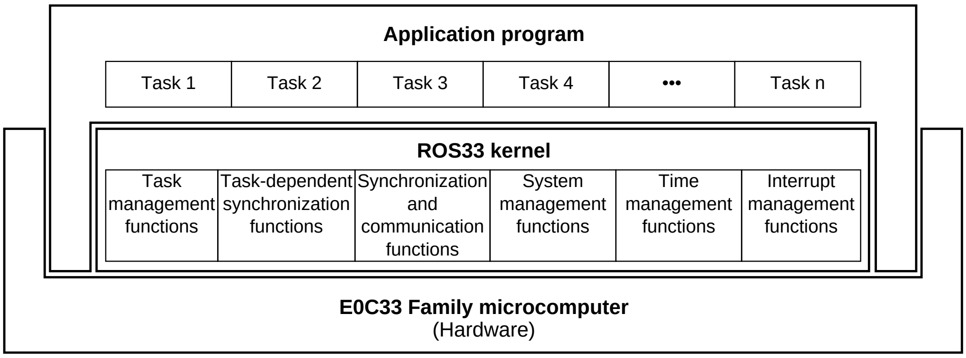


Figure 2.1.1 Conceptual diagram of a system configuration

Functional classification

The functions of the ROS33 kernel are classified into the following six categories:

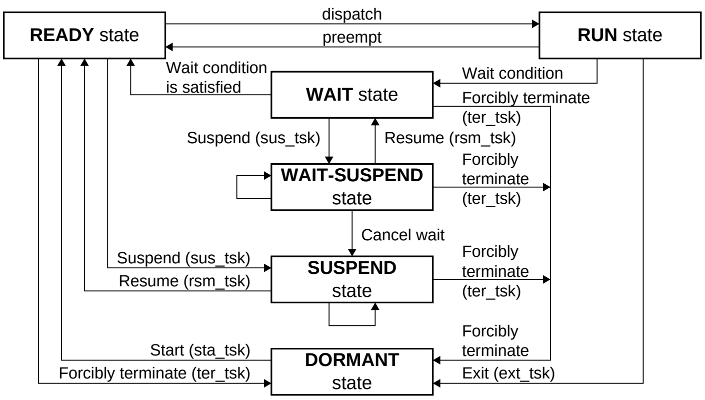
1. Task management functions
   These functions manipulate task states by, for example, starting and terminating a task.
2. Task-dependent synchronization functions
   These functions establish task to task-dependent synchronization by setting or waking up a task to and from a wait state or setting or resuming a task to and from a suspend (forcible wait) state.
3. Synchronization and communication functions
   These functions provide synchronization and communication independently of tasks, issuing and checking events through a semaphore, event flag, and mailbox. A message buffer which is an extended synchronization and communication function.
4. System management functions
   These functions reference the system environment.
5. Time management functions
   These functions set and reference time, and place a task in a wait state for a given time.

6. Interrupt management functions
These functions enable and disable interrupts.

In addition to the above, µITRON 3.0 has several other defined functions—including connection, extended synchronization and communication, memory pool management, and network support functions. However, these functions are only partially supported by ROS33.

Tasks

In ITRON, each unit of parallel processing performed by a program is called a "task". When multiple tasks are started (activated and ready for execution), these tasks are placed in a ready queue (execution wait queue) from which the task with the highest priority is executed. Individual tasks are identified by a numeric value called the "task ID". As task ID values in ROS33 range from 1 to 255, up to 255 tasks can be executed (by default, 8 tasks). Priority is represented by numeric values 1 to 9 (by default, 1 to 8)—the smaller the value, the higher the priority. Tasks with the same priority are executed in the order they have been placed in the ready queue. This order can be changed by a system call, however.
Tasks in executable state are changed over by a system call that causes a transition of task status or by an interrupt. This changeover is called "dispatching". The task under execution can place itself in a wait or halt state, allowing for the task with the next highest priority to be dispatched and placed in executable state. If a task with a higher priority than that of the currently executed task becomes executable, that task is dispatched. The task being executed is returned to an executable state. This is called "preempting".
Figure 2.1.2 shows the transition of task statuses in ROS33.



( ) indicates a system call.

Figure 2.1.2 Transition of task statuses

RUN (execution) state

This state means that the task is currently being executed. This state remains intact until the task is placed in WAIT or DORMANT state or interrupted by an interrupt.

READY (executable) state

This state means that the task has been placed in the ready queue after being started up, or freed from a wait or forcible wait state. The task is currently suspended because some other task with higher priority (or a task with the same priority but placed ahead in the queue) is being executed.

WAIT state

This state means that the task is waiting for an event (message receipt, semaphore acquisition, or event flag setting) or is left suspended due to a system call issued by the task itself. This state remains intact until an event is issued, the task is caused to resume (freed from a wait state) by some other task being executed or by an interrupt handler, or the task is forcibly terminated. In this wait state, semaphore and other resources remain occupied. The resumed task is placed in the ready queue at the end of a queue of tasks with the same priority. After being dispatched, the task has its program counters and registers restored to their previous states at the time of the interruption, and the task begins executing from where it left off.

SUSPEND (forcible wait) state

This state means that task execution has been suspended by a system call from some other task. This state remains intact until the task is restarted by some other task being executed or forcibly terminated. In this wait state, semaphores and other resources remain occupied.
The forcibly suspended task is not erased from the ready queue.

WAIT-SUSPEND (double wait) state

This state is a case where the above WAIT state and SUSPEND state overlap each other. If one of the two wait states is cleared, the task enters the other wait state.

DORMANT state

This state means that the task has not been started yet or has been terminated.
Unlike the wait state, the task relinquishes all resources and accepts no system calls except for startup. When the task restarts executing after startup, its context is initialized.

Task-independent portion

Although the system in almost all cases is placed in a task execution state, it sometimes goes to a non-task execution state, such as for execution of the OS itself. The interrupt handler and timer handler, in particular, are closely tied to the hardware, so they are called "task-independent portions". Task-independent portions are created in the user program along with the tasks.
Task-independent portions (interrupt handler) are executed preferentially over all tasks. When the interrupt handler starts, the tasks currently being executed are suspended, and execution resumes after the interrupt handler is terminated. Also, when the interrupt handler is running, dispatches or any other task transitions are not performed. For example, even if a task is waked up within the interrupt handler and the task has a high enough priority to be dispatched, no dispatching occurs until the interrupt handler is terminated.
Furthermore, a limited number of system calls can be used in task-independent portions.

Interrupt

Interrupts are processed as a task-independent portion, not a task. It is not necessary to define interrupt handlers as tasks.

2.2 List of System Calls

Table 2.2.1, Table 2.2.2, and Table 2.2.3 list the system calls supported by ROS33. For details about each system call, refer to Chapter 3, "System Call Reference".

Table 2.2.1 List of system calls

Classification	System call	Function
Task management	**dis_dsp( )**	Disable Dispatch
	ena_dsp( )	Enable Dispatch
	sta_tsk( )	Start Task
	ext_tsk( )	Exit Issuing Task
	ter_tsk( )	Terminate Other Task
	chg_pri( )	Change Task Priority
	rot_rdq( )	Rotate Tasks on the Ready Queue
	rel_wai( )	Release Wait of Other Task
	get_tid( )	Get Task Identifier
Task-dependent synchronization	**slp_tsk( )**	Sleep Task
	wup_tsk( )	Wake Up Other Task
	sus_tsk( )	Suspend Other Task
	rsm_tsk( )	Resume Suspended Task
	can_wup( )	Cancel Wake Up Request
Synchronization and communication	**wai_sem( )**	Wait on Semaphore
	preq_sem( )	Pall and Request Semaphore
	sig_sem( )	Signal Semaphore
	rcv_msg( )	Receive Message from Mailbox
	prcv_msg( )	Poll and Receive Message from Mailbox
	snd_msg( )	Send Messages to Mailbox
	wai_flg( )	Wait on Event Flag
	pol_flg( )	Wait for Event Flag (Polling)
	set_flg( )	Set Event Flag
	clr_flg( )	Clear Event Flag
System management	**get_ver( )**	Get Version Information
Time management	**set_tim( )**	Set System Clock
	get_tim( )	Get System Clock
	dly_tsk( )	Delay Task
Interrupt management	**loc_cpu( )**	Lock CPU
	unl_cpu( )	Unlock CPU
	ret_int( )	Return from Interrupt Handler
Implementation-dependent functions	**ent_int( )**	Initialize Interrupt Handler Value
	vcre_tsk( )	Create Task
	vcre_mbf( )	Message Buffer Definition
	sys_clk( )	System Clock
	vchg_semcnt( )	Change Semaphore Count Value
	vchk_timer( )	Check Time Management
Extension synchronization and communication	**snd_mbf( )**	Send Messages to Message Buffer
	psnd_mbf( )	Send Messages to Message Buffer (Polling)
	tsnd_mbf( )	Send Messages to Message Buffer (with time-out)
	rcv_mbf( )	Receive Messages from Message Buffer
	prcv_mbf( )	Receive Messages from Message Buffer (Polling)
	trcv_mbf( )	Receive Messages from Message Buffer (with time-out)

Two kinds of system calls described below can be used with the task-independent portions (interrupt handler).

System calls for interrupt handler only

These system calls automatically save or restore the register needed internally. Function of a system call is same as the system call without the prefix “i”.

Table 2.2.2 system call for interrupt handler only

System call	Function
iwup_tsk( )	Wake Up Other Task (wup_tsk, used from the interrupt handler)
iset_flg( )	Set Event Flag (set_flg, used from the interrupt handler)
isig_sem( )	Signal Semaphore (sig_sem, used from the interrupt handler)
isnd_msg( )	Send Messages to Mailbox (snd_msg, used from the interrupt handler)
ipsnd_mbf( )	Send Messages to Message Buffer (psnd_mbf, used from the interrupt handler)

System calls being same as tasks

Use these system calls from an interrupt handler, but be sure to avoid %r15, %alr, and %ahr from %r0 with user’s responsibility assigned before their use.

Table 2.2.3 System calls being same as tasks

System call	Function
wup_tsk( )	Wake Up Other Task
set_flg( )	Set Event Flag
sig_sem( )	Signal Semaphore
snd_msg( )	Send Messages to Mailbox
psnd_mbf( )	Send Messages to Message Buffer (Polling)

2.3 Creating an Application Program

This section describes the precautions to be observed when creating an ROS33 application program by using the program "demo.c" in the "demo¥" directory and sample programs in the "sample¥" directory. For details on how to handle software development tools and how to create C and assembly sources, refer to the "E0C33 Family C Compiler Package Manual".
The following sample programs assume that "ros33.lib" to be linked is generated under the default condition shown on Page 19.

Rules for main function

Shown below is the main function in "demo.c".

Example:

```
#include      "ros33.h"

void main()
{
    sys_ini();

    vcre_tsk(1, task1, 1, (UW)&(stack1[0xa0]));
    vcre_tsk(2, task2, 2, (UW)&(stack2[0xa0]));
    vcre_tsk(3, task3, 2, (UW)&(stack3[0xa0]));
    vcre_tsk(4, task4, 3, (UW)&(stack4[0xa0]));
    vcre_tsk(5, task5, 5, (UW)&(stack3[0xa0]));
    vcre_tsk(8, idle_task, 8, (UW)&(idle_stack[0xa0]));

    sta_tsk(1, 0);
    sta_tsk(2, 0);
    sta_tsk(3, 0);
    sta_tsk(4, 0);
    sta_tsk(8, 0);

    sys_sta();
}
```

In the main function, always be sure to call sys_ini( ) first and sys_sta( ) at the end of the function. The function sys_ini( ) is used to initialize the parameters and resources used by ROS33. After this function, write your user program. In the above example, six tasks are defined by vcre_tsk( ), of which five tasks are started by sta_tsk( ). The last function sys_sta( ) causes the system to start executing in a multitask environment.
Furthermore, "ros33.h" must be included.

Task

All tasks to be executed must be defined using vcre_tsk( ) in the main function. Operation cannot be guaranteed for system calls that use a task ID which is not defined here.
In the example of main( ) above, task1 is defined first.
Example: `vcre_tsk(1, task1, 1, (UW)&(stack1[0xa0]));`

This system call defines the task as task ID = 1 (first argument), task 1 = startup address (second argument), priority = 1 (third argument), and the initial address of the stack used by this task = stak1[ ] (fourth argument). Since this task has priority 1 (the highest priority), when this task is started it is dispatched before any other tasks.
When the tasks are initially defined, they are in DORMANT state. Use sta_tsk( ) to start a task.
Example: `sta_tsk(1, 0);`

The first argument in sta_tsk( ) is a task ID. The second argument is the task startup code (int) to specify the parameter to be passed to the task. However, because ROS33 does not use this code, always specify 0 for the task startup code.

To create each individual task, use the ordinary function format shown below. Note, however, that tasks do not have a return value. Consider the task status transition in Figure 2.1.2 when you create tasks.

Example:

```
void task1( void )
{
     while(1) {
       rcv_msg(&ppk_msg, 1);
       puts(ppk_msg->msgcont);
       slp_tsk();
    }
}
```

This task uses rcv_msg( ) to receive a message from the mailbox and output it. Then the task places itself in WAIT state using slp_tsk( ). This wait state remains effective until the task is waked up by some other task.

If no message exists in the mailbox, task1 is set in a wait state by rcv_msg( ). When a message has been prepared, it is waked up and performs the above processing.

Idle task

An idle task needs to be provided in the user program for times when no tasks are in an executable state. This task must be enabled for interrupt acceptance and must be assigned the lowest priority. It also must always be kept active in main( ). An idle_task is defined in "demo.c".

Example:

```
void idle_task()
{
    while(1){
        for(;;);
    }
}
```

The operation of the OS cannot be guaranteed if the sequence returns from the idle task.

Stack

For the stack, specify a different area for each task. However, for tasks that are not processed in parallel, the same stack area can be shared in order to suppress the amount of RAM spent for tasks. When sharing the stack in this way, make sure that all but one task sharing the stack are in DORMANT state.

In addition to tasks, the system uses about 180 bytes per a task (varies depending on the environment) for the stack for initialization and other purposes. Add this stack to the total amount of stack used by tasks as you allocate the stack area in RAM.

A sample program for sharing a stack is shown below.

Example:

```
#include <stdio.h>
#include "ros33.h"

const char sTask[] = "task";

void main()
{
    sys_ini();

    vcre_tsk(1, task1, 1, (UW)&(stack_common[STACK_SIZE]));
    vcre_tsk(2, task2, 1, (UW)&(stack_common[STACK_SIZE]));
    vcre_tsk(3, task_main, 2, (UW)&(stack_main[STACK_SIZE]));
    vcre_tsk(8, idle_task, 8, (UW)&(stack_idle[STACK_SIZE]));

    /* start idle task */
    sta_tsk(8, 0);

    /* start main task */
    sta_tsk(3, 0);

    sys_sta();
}

void task_main( void )
{
    sta_tsk(1, 0);
    sta_tsk(2, 0);
    slp_tsk();
}
```

```
void task1(void)
{
    char str[10];
    strcpy(str, sTask);
    strcat(str, "1");
    puts(str);
    ext_tsk();
}
void task2(void)
{
    char str[10];
    strcpy(str, sTask);
    strcat(str, "2");
    puts(str);
    slp_tsk();
}
```

1. The same stack area is defined for both task1 and task2 using the vcre_tsk() system call.
2. task_main() enters RUN state by sys_sta() in the main function.
3. task1 enters RUN state by sta_tsk(1,0) in the main function.
4. task1 enters DORMANT state by ext_tsk(), then task_main() enters RUN state.
5. task2 enters RUN state by sta_tsk(2,0) in task_main ().
6. task2 enters WAIT state by slp_tsk(), then task_main() enters RUN state.

In this example, task1 and task2 use the same stack area. Since task1 and task 2 do not enter the same state other than DORMANT state, stack sharing is possible.

For reference, a sample source for stack sharing is provided in the "sample¥" directory.

Initializing the dispatcher

The task dispatcher uses software exception 0.
Register int_dispatch to the corresponding vector address.

Interrupt

Create an interrupt handler for each factor of interrupts used in your application, and write its start address to the corresponding interrupt vector address. When the interrupt factor is generated, the corresponding interrupt handler is executed as a task-independent portion. The tasks that have until now been executed are suspended from execution until the interrupt handler completes its processing. Also, the E0C33 chip's trap processing is initiated and the interrupts whose priority levels are below that of the interrupt being serviced are masked out during this time. To enable multiple interrupts, directly set the IE bit of the PSR. For details about interrupts, refer to the Technical Manual supplied with each E0C33 Family microcomputer.

Interrupt handler

1. After an interrupt occurs, generate ent_int( ) before permitting a CPU interrupt (sets PSR IE bit at 1) or generating a system call.
   ent_int( ) is a system call which notifies the system of the interrupt process.
2. Clear the interrupt factor flag.
3. Execute the user’s interrupt processing or system call. If there is a register needed for user interrupt process, save to the stack.
4. Restore contents of the registers which have been saved to the stack.
5. Issue ret_int( ) and end the interrupt handler. Do not end with the “reti” command.

- The interrupt handler uses a stack of tasks which were being executed until now. If you want to designate a stack exclusively for the interrupt handler, switch %sp immediately after starting the interrupt handler and immediately before terminating it.
- Set the values of registers %r0 to %r15, %alr, %ahr, and %sp determined right before issuing ret_int to the same values just after the interrupt occurs.
- It is not necessary to issue ent_int and ret_int system calls when multiple interrupts or NMI is not used, and when OS system call is not issued from the interrupt handler.

Interrupt handler sample:

```
int_hdr:
    xcall     ent_int           ; Start interrupt handler

    pushn     %r15              ; Save register
    ld.w      %r1,%alr
    ld.w      %r0,%ahr
    pushn     %r1

    xld.w     %r0,IFCT_TM160    ; Clear interrupt factor flag.
    ld.w      %r1,1
    ld.w      [%r0],%r1

    xcall     usr_routine       ; Execute user routine.

    ld.w      %r12,0x1
    xcall     wup_tsk           ; Generate wup_tsk(1).

    popn      %r1               ; Restore registers.
    ld.w      %alr,%r1
    ld.w      %ahr,%r0
    popn      %r15

    xcall     ret_int           ; End interrupt handler.
```

System call issued from interrupt handler

There are two types of system calls generated from an interrupt handler: exclusive system calls and system calls identical to tasks.

● System calls exclusively for interrupt handler

These system calls execute automatically to save and restore the minimum numbers of registers required internally. This is faster than using system calls identical to tasks, and the stack size is most optimal.

iwup_tsk	Wakes up another task. (Use from interrupt handler. wup_tsk)
iset_flg	Sets an event flag. (Use from interrupt handler. set_flg)
isig_sem	Returns a semaphore resource. (Use from interrupt handler. sig_sem)
isnd_msg	Sends a message to the mailbox. (Use from interrupt handler. snd_msg)
ipsnd_mbf	Sends a message to the massage buffer. (Use from interrupt handler. psnd_mbf)

Example：

```
interrupt:
    xcall     ent_int           ; Start interrupt handler.

    sub       %sp, 2            ; Save %r10, %r12
    ld.w      [%sp+0], %r10     ; %r10 = store iwup_tsk return value
    ld.w      [%sp+1], %r12     ; %r12 = store iwup_tsk parameter

    ld.w      %r12,0x1
    xcall     iwup_tsk          ;  Generate wup_tsk(1)

    ld.w      %r10,[%sp+0]
    ld.w      %r12,[%sp+1]
    add       %sp, 2            ; Return %r10, %r12.

    xcall     ret_int           ; End interrupt handler.
```

● System calls identical to tasks

These system call do not execute to save and return the minimum numbers of registers internally. Before using them, be sure to have the user save %r0 - %r15, %alr, and %ahr. Use these system calls when saving all registers at the start of an interrupt. (Use the C language to write the interrupt handler.) In this case, when you use a system call for exclusive interrupt handler, note that stack is used excessively.

wup_tsk	Wake up another task.
set_flg	Sets event flag.
sig_sem	Signal semaphore.
snd_msg	Sends a message to the mailbox
psnd_mbf	Sends a message to the message buffer (polling)

Example :

```
int_hdr:
     xcall    ent_int            ; Start interrupt handler

     pushn    %r15               ; Save register.
     ld.w     %r1,%alr
     ld.w     %r0,%ahr
     pushn    %r1                ;

     xcall    usr_routine        ; Execute user routine written in C language

     ld.w     %r12,0x1
     xcall    wup_tsk            ; Generate wup_tsk(1).

     popn     %r1                ; Restore registers.
     ld.w     %alr, %r1
     ld.w     %ahr, %r0
     popn     %r15

     xcall    ret_int            ; End interrupt handler
```

Interrupt level in system call

An interrupt occurring while a system call is being executed is masked by IL of the CPU (interrupt level). While the system call is being executed, the CPU interrupt level is set to default 4.
For interrupts with the following priority, executable system calls can be run from the interrupt handler. For interrupts with priority exceeding this setting, the system calls cannot be executed.

With this default setting, a system call is possible from interrupt of level 0, 1, 2, 3, and 4; but not possible from interrupt of level 5, 6, and 7. An interrupt where system call is not possible is not related to the OS and enables interrupt processing with good response. This interrupt does not need to use ent_int and ret_int. System call cannot be generated from this interrupt handler. Also, if the interrupt occurs multiple times during interrupt handler processing and system call is generated, the operation is not guaranteed. To change the interrupt mask level in a system call, have the user perform this statically during the ROS33 configuration. The setting can be made by changing the two locations in the source file, as follows:

● Changing the command (which operates the PSR register of the mDisableInt macro) at around line 100 in src¥internal.h

```
asm("
     ld.w %r9, %psr
     xand %r9, %r9, 0xff        ; clear IL
     xoor %r9, %r9, 0x400       ; set IL   <- Interrupt level 4
     ld.w %psr, %r9
");
```

Default is level 4. To switch to level 5, use the 3rd command line as follows:

```
xoor %r9, %r9, 0x500     ; set IL     <- Interrupt level 5
```

- Changing the command (which operates the PSR register of the macro) at around line 20 in src¥ros33asm.s.

```
#macro ENTER_CRITICAL_SECTION
    pushn %r0
    ld.w  %r0, %psr
    xand  %r0, %r0, 0xff        ; clear IL
    xoor  %r0, %r0, 0x410       ; set IL and IE, interrupt mask by IL
    ld.w  %psr, %r0
    popn  %r0
#endm
```

Default is level 4. To switch to level 5, , use the 3rd command line as follows:

xoor %r0, %r0, 0x510 ; set IL <- Interrupt level 5

Timer handler

When using time management function system calls (set_tim, get_tim, dly_tsk), create a timer handler in the user program that calls sys_clk( ) every 1 ms. Normally, use a 16-bit timer to generate an interrupt every 1 ms and then make it an interrupt handler.

Example:

"timer_hdr" symbol is registered for timer vector

```
.global timer_hdr
timer_hdr:
    xcall   ent_int     ; Start interrupt handler.
    pushn   %r1         ; Save register.
    xld.w   %r1,0x40    ; Clear timer factor flag.
    xld.w   %r0,0x40284
    ld.w    [%r0],%r1
    popn    %r1         ; Restore registers.
    xcall   sys_clk     ; Generate sys_clk.
    xcall   ret_int     ; End interrupt handler.
```

For your reference, a sample source including 16-bit timer setting is provided in the “sample¥” directory.

Usage example of a mailbox

```
#include <stdio.h>
#include "ros33.h"

T_MSG msg;

void task1( void )
{
   T_MSG* pk_msg;

   while(1) {
      rcv_msg(&pk_msg, 1);
      puts(pk_msg->msgcont);
      slp_tsk();
   }
}

void task2( void )
{
   while(1) {
      strcpy(msg.msgcont, "HELLO");
      msg.pNxt = 0;   /* message init */
      snd_msg(1, &msg);
      slp_tsk();
   }
}
```

This sample program assumes that task1 and task2 are placed in the same ready queue with a priority level in the order of task1 and task2, and there is no message in the mailbox (ID1).

1. task1 enters RUN state. The rcv_msg() in task1 requests to receive a message. task1 enters WAIT state since the mailbox (ID1) has no message.
2. task2 enters RUN state. task1 initializes a message and sends it to the mailbox (ID1) using snd_msg(). This makes task1 enter READY state.
3. task1 enters RUN state by slp_tsk() in task2.
4. task1 outputs the received message.

For reference, a sample source that uses a mailbox is provided in the "sample¥" directory.

Message structure:

The message structure T_MSG is defined in "itron.h" as follows:

```
typedef struct t_msg {
  struct t_msg*  pNxt;          ... Message header
  VB             msgcont[10];   ... Message body
} T_MSG;
```

A message consists of a header (first 4 bytes) and a message body.
To expand a message body into 10 bytes or more, define as follows:
Example:

```
VB      msg_buf[25];
T_MSG*  pk_msg;
pk_msg = (T_MSG*)msg_buf;
```

The message header (pNxt) must be initialized to 0 before using the massage.

Usage example of a semaphore

```
void task1( void )
{
   while(1) {
      wai_sem(1);
      rot_rdq(1);
      sig_sem(1);
      puts("task1");
      slp_tsk();
   }
}
void task2( void )
{
   while(1) {
      wai_sem(1);
      puts("task2");
      sig_sem(1);
      slp_tsk();
   }
}
```

This sample program assumes that task1 and task2 are placed in the same ready queue with a priority level in the order of task1 and task2, and the resource of the semaphore (ID1) has not be returned.

1. task1 enters RUN state and gets the resource from the semaphore (ID1) using wai_sem().
2. task2 enters RUN state by rot_rdq() in task1.
3. task2 requests the resource from the semaphore (ID1). task2 enters WAIT state since it cannot get the resource.
4. task1 enters RUN state and returns the resource to the semaphore (ID1) using sig_sem(). This makes task2 enter READY state.
5. task2 enters RUN state by slp_tsk() in task1.

For reference, a sample source that uses a semaphore is provided in the "sample¥" directory.

Usage example of an event flag

```
#include <stdio.h>
#include "ros33.h"
void task1( void )
{
   UINT p_flgptn;

   while(1) {
      wai_flg(&p_flgptn, 1, 0x11, TWF_ANDW);
      printf("Flag pattern 0x%x¥n", p_flgptn);
      slp_tsk();
   }
}
void task2( void )
{
   while(1) {
      set_flg(1, 0x11);
      slp_tsk();
   }
}
```

This sample program assumes that task1 and task2 are placed in the same ready queue with a priority level in the order of task1 and task2, and the event flag (ID1) has be set to 0x00.

1. task1 enters RUN state. task1 enters WAIT state after executing wai_flag() that waits for the event flag (ID1) to be set to the specified status.
2. task2 enters RUN state and sets the event flag (ID1) to 0x11 using set_flg(). Since this releases the flag waiting condition for task1, task1 enters READY state.
3. task1 enters RUN state by slp_tsk() in task2.
   task2 outputs the contents of the event flag that has been released from the waiting condition using printf().

For reference, a sample source that uses an event flag is provided in the "sample¥" directory.

Usage example of a message buffer

```
#include "ros33.h"

#define BUFSIZE 20
#define MAX_MSG_SIZE 15
char buf1[BUFSIZE];

void   main()
{
   sys_ini();

   ...

   vcre_mbf(1, buf1, BUFSIZE, MAX_MSG_SIZE);  /* create message buffer */

   ...

   sys_sta();                             /* start main task */
}

void task1( void )
{
   char tbuf[MAX_MSG_SIZE];
   int  tsize;

   while(1) {
      rcv_mbf(tbuf, &tsize, 1);
      slp_tsk();
   }
}

void task2( void )
{
   while(1) {
      snd_mbf(1, "TASK1 Message", 14);
      slp_tsk();
   }
}
```

This sample program assumes that task1 and task2 are placed in the same ready queue with a priority level in the order of task1 and task2, and the message buffer (ID1) has no message.

1. Define the message buffer(ID1). Use vcre_mbf() function between sys_ini() and sys_sta() in the main function. The system call operation can not be guaranteed with the message buffer ID which is not defined in this section.
2. task1 enters RUN state. task1 enters WAIT state after initializing a message and sending it in task1 rcv_mbf() because no message is found in the message buffer(ID1).
3. task2 enters RUN state. By snd_mbf(), a message is sent to the message buffer(ID1). This makes task1 enter READY state. The message is copied to tbuf.

For reference, a sample source that uses a message buffer is provided in the "sample¥" directory.

Building an application program

The ROS33 modules are provided as the library file "ros33.lib" in the "lib¥" directory. Link this library with the user modules. When linking, specify the said directory as a library path in the linker command file. Only those modules required for the system calls used will be linked.

Example:
```
;Library path
-l C:¥CC33¥lib        ....CC33 standard library
-l C:¥ROS33¥lib       ....ROS33 standard library
```

Note that "ros33.lib" is created as a standard kernel that includes an error check function but omits debug functions. If you want to change this function or the maximum resource value, customize the library as necessary. (Refer to Section 2.4, "Customizing ROS33".)

Precautions

- All tasks to be executed must be defined in the main function by using vcre_tsk( ). Operation cannot be guaranteed for system calls that use an undefined task ID.
- The idle task must be enabled for interrupt acceptance and must be assigned the lowest priority. Furthermore, do not return from the idle task.
- To enable or disable interrupts in tasks, always be sure to use system calls loc_cpu( ) or unl_cpu( ). Operation cannot be guaranteed if PSR is changed by operating on it directly.
- The stack for each task should be prepared with an enough size.
- To enable multiple interrupts in an interrupt handler, directly set the IE (interrupt enable) bit of the PSR.
- Most system calls are written in C. If you use the assembly language in the development, be sure to consider the register layout . For details, refer to the C compiler package manual.

2.4 Customizing ROS33

The library "ros33.lib" is created with the following features:

Resources

Resources		(Valid setup range)
Number of tasks	8	(1 to 255)
Priority levels	8	(1 to 9)
Number of event flags	8	(0 to 255)
Number of semaphores	8	(0 to 255)
Number of message buffers	8	(0 to 255)
Number of mailboxes	8	(0 to 255)
Semaphore count value	1	(1 to 255)
Wakeup count value	1	(1 to 255)
Initial value of PSR	0x00000010	...Interrupt enabled
Timer handler interrupt cycle	1	(multiples of 8)

Compile options

NO_ERROR_CHECK option	Unspecified
DEBUG_KERNEL option	Unspecified
NO_RETURN_VALUE option	Unspecified
USE_GP option	Unspecified

The ROS33 source files are provided in the "src¥" directory, so you can customize it following the procedure described below.

Method for changing resources

The maximum value of each resource and the initial value of PSR are defined in "include¥ros33.h". Change the contents of these definitions as necessary, then recompile the file.

Contents of definitions in "ros33.h"

```
// If you change resource number please edit following.
#define SMPH_NUM    8                    // max semaphore, 0 to 255
#define FLG_NUM     8                    // max flag, 0 to 255
#define MLBX_NUM    8                    // max mailbox, 0 to 255
#define MSGBUF_NUM  8                    // max message buffer, 0 to 255
#define TSK_NUM     8                    // max task, 1 to 255

#define MAX_TSKPRI  8                    // max task priority, 1 to 9

#define SMPH_CNT    1                    // semaphore count, 1 to 255
#define WUP_CNT     1                    // max wakeup count 1 to 255

#define INI_PSR     0x00000010           // initial flag (%PSR value)
                                         // default is interrupt enable
#define INT_TIME    1                    // timer interrupt time (ms)
```

Note on INT_TIME

Set it up so that the timer handler calling sys_clk does not occur every 1m sec. Make sure that the number is multiple of 8, for example 8 for 8m sec and 16for 16 sec.

Compile options and recompilation

Be sure to compile using the "-0" option which allows for optimum GCC33. For the complier, use gcc33 Ver2.7.2.

NO_ERROR_CHECK option

By compiling the file after specifying "-DNO_ERROR_CHECK" with a gcc33 startup command, you can generate a very compact kernel with error check functions omitted. However, because occurrence of an error causes the system to crash, this option can only be used when you are absolutely certain that no errors will occur.

DEBUG_KERNEL option

By specifying "-DDEBUG_KERNEL" with a gcc33 startup command and "-d DEBUG_KERNEL" with a pp33 startup command, you can generate a debug kernel. When a debug kernel is generated, the dispatcher (a functional block to control dispatch in the OS) has an added function. This function calls two other functions, which are described below:

`void ros_dbg_tskcng(ID tskid)`

This function is called when the task to be dispatched has been confirmed.

`void ros_dbg_stackerr()`

This function is called when an error occurs in the stack used by a task being executed.
If the task stack area is used to exchange messages with the mailbox, the system accesses the stack for the task being executed, which causes a stack error.

Note that these functions are not included in ROS33. Therefore, they need to be created in the user program. For your reference, examples of these functions are provided in "src¥debug.c".

NO_RETURN_VALUE option

By specifying "-DNO_RETURN_VALUE" with a gcc33 startup command, a compact kernel that has no function to set return values can be generated. In this case, system calls do not set any return value, so undefined values will be returned.

USE_GP option

If you want to optimize the code using a global pointer, change the address at which the global pointer definition is defined in "ros33.h" to your desired address and specify "-DUSE_GP" with a gcc33 startup command before compiling "tskmng.c."

Global pointer definition in "ros33.h"

```
// If you use global pointer please edit here
#ifdef USE_GP
#define GLOBAL_POINTER  0x00000000  // global pointer (%r8 value)
#endif
```

Note that a make file to generate "ros33.lib" has been created in the "build¥" directory. Recompile the file after modifying necessary points.

"ros33.mak"

```
# macro definitions for tools & dir

TOOL_DIR = C:¥CC33
GCC33 = $(TOOL_DIR)¥gcc33
PP33  = $(TOOL_DIR)¥pp33
EXT33 = $(TOOL_DIR)¥ext33
AS33  = $(TOOL_DIR)¥as33
LK33  = $(TOOL_DIR)¥lk33
MAKE  = $(TOOL_DIR)¥make
LIB33 = $(TOOL_DIR)¥lib33
DEBUG = -g
SRC_DIR = ..¥src¥¥

# macro definitions for tool flags

#for release kernel (error check)
GCC33_FLAG = -B$(TOOL_DIR)¥ $(DEBUG) -S -I..¥include -O
PP33_FLAG  = $(DEBUG)

#for debug kernel
#GCC33_FLAG = -B$(TOOL_DIR)¥ $(DEBUG) -S -I..¥include -O -DDEBUG_KERNEL
#PP33_FLAG  = -d DEBUG_KERNEL $(DEBUG)

#for release kernel (NO error check)
#GCC33_FLAG = -B$(TOOL_DIR)¥ $(DEBUG) -S -I..¥include -O -DNO_ERROR_CHECK
#PP33_FLAG  = $(DEBUG)
```

```
EXT33_FLAG =
AS33_FLAG  = $(DEBUG)

# suffix & rule definitions

.SUFFIXES : .c .s .ps .ms .o .srf

.c.ms :
     $(GCC33) $(GCC33_FLAG) $(SRC_DIR)$*.c
     $(EXT33) $(EXT33_FLAG) $*.ps

.s.ms :
     $(PP33)  $(PP33_FLAG)  $(SRC_DIR)$*.s
     $(EXT33) $(EXT33_FLAG) $*.ps

.ms.o :
     $(AS33)  $(AS33_FLAG)  $*.ms

# dependency list

ros33.lib : flag.o intmng.o mailbox.o ros33.o ros33asm.o semapho.o timemng.o ¥
 tskmng.o tsksync.o debug.o
     $(LIB33) -a ros33.lib  flag.o intmng.o mailbox.o ros33.o ros33asm.o ¥
semapho.o timemng.o tskmng.o tsksync.o debug.o
     copy ros33.lib ..¥lib
     del ros33.lib
         .
         .
         .
```

3 System Call Reference

This section explains the functions of each system call.

3.1 List of System Calls

Table 3.1.1, Table 3.1.2, and Table 3.1.3 list the system calls supported by ROS33.

Table 3.1.1 List of system calls

Classification	System call	Function
Task management	**dis_dsp**(void)	Disable Dispatch
	ena_dsp(void)	Enable Dispatch
	sta_tsk(ID tskid, INT stacd)	Start Task
	ext_tsk(void)	Exit Issuing Task
	ter_tsk(ID tskid)	Terminate Other Task
	chg_pri(ID tskid, TPRI tskpri)	Change Task Priority
	rot_rdq(TPRI tskpri)	Rotate Tasks on the Ready Queue
	rel_wai(ID tskid)	Release Wait of Other Task
	get_tid(ID *p_tskid)	Get Task Identifier
Task-dependent synchronization	**slp_tsk**(void)	Sleep Task
	wup_tsk(ID tskid)	Wake Up Other Task
	sus_tsk(ID tskid)	Suspend Other Task
	rsm_tsk(ID tskid)	Resume Suspended Task
	can_wup(INT *p_wupcnt, ID tskid)	Cancel Wake Up Request
Synchronization and communication	**wai_sem**(ID semid)	Wait on Semaphore
	preq_sem(ID semid)	Pall and Request Semaphore
	sig_sem(ID semid)	Signal Semaphore
	rcv_msg(T_MSG **ppk_msg, ID mbxid)	Receive Message from Mailbox
	prcv_msg(T_MSG **ppk_msg, ID mbxid)	Poll and Receive Message from Mailbox
	snd_msg(ID mbxid, T_MSG *pk_msg)	Send Messages to Mailbox
	wai_flg(UINT *p_flgptn, ID flgid, UINT waiptn, UINT wfmode)	Wait for Event Flag
	pol_flg(UINT *p_flgptn, ID flgid, UINT waiptn, UINT wfmode)	Wait for Event Flag (Polling)
	set_flg(ID flgid, UINT setptn)	Set Event Flag
	clr_flg(ID flgid, UINT clrptn)	Clear Event Flag
System management	**get_ver**(T_VER *pk_ver)	Get Version Information
Time management	**set_tim**(SYSTIME *pk_tim)	Set System Clock
	get_tim(SYSTIME *pk_tim)	Get System Clock
	dly_tsk(DLYTIME dlytim)	Delay Task
Interrupt management	**loc_cpu**(void)	Lock CPU
	unl_cpu(void)	Unlock CPU
	ret_int(void)	Return from Interrupt Handler
Implementation-dependent	**ent_int**(void)	Initialize Interrupt Handler Value
	vcre_tsk(ID tskid, FP task, PRI itskpri, UW istkadr)	Create Task
	vcre_mbf(ID mbfid, VP msgbuf, INT bufsz, INT maxmsz)	Create Message Buffer
	sys_clk( )	System Clock
	vchg_semcnt(ID semid, UB semcnt)	Change Semaphore count value
	vchk_timer( )	Check Time Management
Extension synchronization and communication	**snd_mbf**(ID mbfid, VP msg, INT msgsz)	Send Messages to Message Buffer
	psnd_mbf(ID mbfid, VP msg, INT msgsz)	Send Messages to Message Buffer (Polling)
	tsnd_mbf(ID mbfid, VP msg, INT msgsz, TMO tmout)	Send Messages to Message Buffer (with time-out)
	rcv_mbf(VP msg, INT *p_msgsz, ID mbfid)	Receive Messages from Message Buffer
	prcv_mbf(VP msg, INT *p_msgsz, ID mbfid)	Receive Messages from Message Buffer (Polling)
	trcv_mbf(VP msg, INT *p_msgsz, ID mbfid, TMO tmout)	Receive Messages from Message Buffer (with time-out)

Two kinds of System call below can be used with the task-independent portions (interrupt handler).

System call for the interrupt handler only

These system call automatically avoid or restore the register needed for the inside. The function of the system call equals to the system call without a prefix “i”.

Table 3.1.2 system call for interrupt handler only

System call	Function
iwup_tsk( )	Wake Up Other Task (wup_tsk, used from the interrupt handler)
iset_flg( )	Set Event Flag (set_flg, used from the interrupt handler)
isig_sem( )	Signal Semaphore (sig_sem, used from the interrupt handler)
isnd_msg( )	Send Messages to Mailbox (snd_msg, used from the interrupt handler)
ipsnd_mbf( )	Send Messages to Message Buffer (psnd_mbf, used from the interrupt handler)

System call same as task

Use this system call from interrupt handler, but be sure to avoid %r15, %alr, and %ahr from %r0 with user’s responsibility before use.

Table 3.1.3 system call same as task

System call	Function
wup_tsk( )	Wake Up Other Task
set_flg( )	Set Event Flag
sig_sem( )	Signal Semaphore
snd_msg( )	Send Messages to Mailbox
psnd_mbf( )	Send Messages to Message Buffer (Polling)

3.2 List of Data Types

Table 3.2.1 lists the data types used for the arguments of each system call.

Table 3.2.1 List of data types

Type	Definition		Description
B	`typedef char`	`B;`	Signed 8-bit integer
H	`typedef short`	`H;`	Signed 16-bit integer
W	`typedef long`	`W;`	Signed 32-bit integer
UB	`typedef unsigned char`	`UB;`	Unsigned 8-bit integer
UH	`typedef unsigned short`	`UH;`	Unsigned 16-bit integer
UW	`typedef unsigned long`	`UW;`	Unsigned 32-bit integer
VW	`typedef long`	`VW;`	Unpredictable data type (32-bit size)
VH	`typedef short`	`VH;`	Unpredictable data type (16-bit size)
VB	`typedef char`	`VB;`	Unpredictable data type (8-bit size)
***VP**	`typedef void`	`*VP;`	Pointer to an unpredictable data type
***FP**	`typedef void`	`(*FP)();`	Program start address
INT	`typedef int`	`INT;`	Signed 32-bit integer
UINT	`typedef unsigned int`	`UINT;`	Unsigned 32-bit integer
BOOL	`typedef H`	`BOOL;`	Boolean value: TRUE (1) or FALSE (0)
FN	`typedef short`	`FN;`	Maximum 2 bytes of function code
ID	`typedef INT`	`ID;`	Object ID number (signed 16-bit integer)
BOOL_ID	`typedef INT`	`BOOL_ID;`	Boolean value or ID number (signed 16-bit integer)
HNO	`typedef INT`	`HNO;`	Handler number (signed 16-bit integer)
ATR	`typedef UINT`	`ATR;`	Object or handler attribute (unsigned 16-bit integer)
ER	`typedef INT`	`ER;`	Error code (signed 16-bit integer)
PRI	`typedef INT`	`PRI;`	Task priority (signed 16-bit integer)
TMO	`typedef INT`	`TMO;`	Timeout value (signed 16-bit integer)
DLYTIME	`typedef TMO`	`DLYTIME;`	Delay time (signed 16-bit integer)

These data types are defined in "include¥itron.h".

3.3 List of Error Codes

Table 3.3.1 lists the error codes returned by system calls.

Table 3.3.1 List of error codes

Error code	Value	Description
E_OK	0	Normal completion
E_SYS	(-5)	System error
E_NOMEM	(-10)	Insufficient memory
E_NOSPT	(-17)	Feature not supported
E_INOSPT	(-18)	Feature not supported by ITRON/FILE specification
E_RSFN	(-20)	Reserved function code number
E_RSATR	(-24)	Reserved attribute
E_PAR	(-33)	Parameter error
E_ID	(-35)	Invalid ID number
E_NOEXS	(-52)	Object does not exist
E_OBJ	(-63)	Invalid object state
E_MACV	(-65)	Memory access disabled or memory access violation
E_OACV	(-66)	Object access violation
E_CTX	(-69)	Context error
E_QOVR	(-73)	Queuing or nesting overflow
E_DLT	(-81)	Object being waited for was deleted
E_TMOUT	(-85)	Polling failure or timeout exceeded
E_RLWAI	(-86)	WAIT state was forcibly released

These error codes are defined in "include¥itron.h".

3.4 Details of System Calls

3.4.1 System Calls of Task Management Functions

Disable Dispatch — dis_dsp

Format: `ER dis_dsp( void );`

Parameter: None

Return values:

`E_OK`	Terminated normally
`E_CTX`	Context error (issued after loc_cpu has been executed from a task-independent portion)

Description: This system call disables task dispatches. From this time onward until ena_dsp is issued, a task itself will never be preempted from RUN state to READY state, though there is a possibility of other tasks with higher priority being placed in READY state. The task is also disabled from entering WAIT or DORMANT state. External interrupts are not disabled, however.

Enable Dispatch — ena_dsp

Format: `ER ena_dsp( void );`

Parameter: None

Return values:

`E_OK`	Terminated normally
`E_CTX`	Context error (issued after loc_cpu has been executed from a task-independent portion)

Description: This system call reenables a dispatch that has been disabled by dis_dsp. If a task with higher priority than the reenabled task itself exists in the ready queue, this task is dispatched at that point in time and the reenabled task is preempted.
If both interrupt and dispatch are disabled by loc_cpu, dispatch is not enabled by this system call and error code E_CTX is returned.
If this system call is issued when dispatch is already enabled, the system call is ignored and no error is assumed.

Start Task — sta_tsk

Format: `ER sta_tsk(ID tskid, INT stacd);`

Parameters:

`ID tskid`	Task ID number
`INT stacd`	Task start code (not used in the system call)

Return values:

`E_OK`	Terminated normally
`E_ID`	Illegal ID number (tskid is illegal or cannot be used)
`E_NOEXS`	Specified task does not exist.
`E_OBJ`	Specified task is not in DORMANT state.

Description: This system call starts the task indicated by tskid. The specified task is registered in the ready queue, and its state is changed from DORMANT to READY. In the ready queue, it is positioned at the end of the queue of tasks with the same priority.
If the specified task has the highest priority among the executable (READY) tasks and there is no other task with the same priority, the task is dispatched and placed in RUN state. In this case, the task being executed when it issued sta_tsk is made the task to be executed next at this time.
Task startup is effective for only those in DORMANT state. If you specify a task in any other state, the task status is not changed and error code E_OBJ is returned.
The second argument "stacd" is not used in ROS33, so specify 0 for it.

Note: Before you can start a task, you must first issue the vcre_tsk system call to define that task.

Exit Issuing Task ext_tsk

Format: `void ext_tsk( void );`

Parameter: None

Return value: None

Description: This system call terminates the task itself that issues this call. The terminated task is placed in an DORMANT state. At the same time, the task with the highest priority in the ready queue is dispatched and placed in RUN state. Use the sta_tsk system call to restart a task that has been terminated by this system call.

Terminate Other Task ter_tsk

Format: `ER ter_tsk( ID tskid );`

Parameter: `ID tskid` Task ID number

Return values:

`E_OK`	Terminated normally
`E_ID`	Illegal ID number (tskid is illegal or cannot be used)
`E_NOEXS`	Specified task does not exist.
`E_OBJ`	Specified task is in DORMANT state or the issuing task itself is specified.

Description: This system call forcibly terminates the task specified by tskid. The terminated task is placed in DORMANT state. If you specify the issuing task itself or a task in DORMANT state, error code E_OBJ is returned. Use the sta_tsk system call to restart a task that has been terminated by this system call.

Change Task Priority chg_pri

Format: `ER chg_pri( ID tskid, TPRI tskpri );`

Parameters:

`ID tskid`	Task ID number
`TPRI tskpri`	Task priority

Return values:

`E_OK`	Terminated normally
`E_ID`	Illegal ID number (tskid is illegal or cannot be used)
`E_NOEXS`	Specified task does not exist.
`E_PAR`	Parameter error (tskpri is illegal or has an unusable value)
`E_OBJ`	Specified task is in DORMANT state.

Description: This system call changes the current priority of the task specified by tskid to a value specified by tskpri. The priority of any task in DORMANT (inactive) state cannot be changed. If an inactive task is specified, error code E_OBJ is returned.
The priority changed here remains effective until the task enters DORMANT state. When the task is placed in DORMANT state, the task's initial priority value set by vcre_tsk is restored.
If the priority of a task in the ready queue is changed, the task is moved to the last position in the task queue with the same priority as its changed priority. This modification is also used to specify the same priority for a task as its current priority, or change the priority of the issuing task itself.

Rotate Tasks on the Ready Queue rot_rdq

Format: `ER rot_rdq( TPRI tskpri );`

Parameter: `TPRI tskpri` Task priority

Return values:

`E_OK`	Terminated normally
`E_PAR`	Parameter error (tskpri is illegal)

Description: This system call rotates a ready queue that has priorities specified by tskpri. The task at the top of the queue with the specified priority is moved to the last position in the queue. In this system call, you can use TPRI_RUN (priority of the task being executed) for tskpri, so that it is possible to rotate the queue that includes the issuing task itself.
If the task of a specified priority (valid value) does not exist in the ready queue, this system call is ignored.
This system call only affects the task queue with the specified priority, and no other task queue is affected.

Release Wait of Other Task rel_wai

Format: `ER rel_wai( ID tskid );`

Parameter: `ID tskid` Task ID number

Return values:

`E_OK`	Terminated normally
`E_ID`	Illegal ID number (tskid is illegal or cannot be used)
`E_NOEXS`	Specified task does not exist.
`E_OBJ`	Specified task is not in a wait state (including the issuing task itself and those in DORMANT state).

Description: If the task specified by tskid is in WAIT state, this system call forcibly frees it (not including SUSPEND state). Error E_RLWAI is returned for the task freed from wait state by rel_wai. This can be used for time-out processing of tasks in a wait state. If the specified task is in WAIT-SUSPEND state, only the WAIT state is cleared and the task goes to SUSPEND state.
If the specified task is neither in WAIT state nor in WAIT-SUSPEND state, error code E_OBJ is returned to the task that had issued this system call.

Get Task Identifier get_tid

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

Parameter: `ID *p_tskid` Pointer to task ID number

Return values:

`E_OK`	Terminated normally
`FALSE=0`	Executed from a task-independent portion

Description: This system call returns the ID number of the issuing task itself. When this system call is issued from a task-independent portion, FALSE = 0 is returned as the task ID.

3.4.2 System Calls of Task-Dependent Synchronization Functions

Sleep Task slp_tsk

Format: `ER slp_tsk( void );`

Parameter: `ID tskid` Task ID number

Return values:

`E_OK`	Terminated normally
`E_RLWAI`	Wait state forcibly cleared (rel_wai accepted during wait state)
`E_CTX`	Context error (executed from a task-independent portion or when dispatch is disabled)

Description: This system call moves the issuing task itself from RUN state to WAIT state. This wait state is cleared by a wup_tsk system call from another task. The wait state also is forcibly cleared when rel_wai is executed by some other task, in which case error code E_RLWAI is returned.
If sus_tsk is executed by some other task, the task is placed in WAIT-SUSPEND state.

Wake Up Other Task wup_tsk

Format: `ER wup_tsk( ID tskid );`

Parameter: `ID tskid` Task ID number

Return values:

`E_OK`	Terminated normally
`E_ID`	Illegal ID number (tskid is illegal or cannot be used)
`E_NOEXS`	Specified task does not exist.
`E_OBJ`	Specified task is the issuing task itself or in DORMANT state.
`E_QOVR`	Wakeup requests exceed the allowable range.

Description: This system call causes a task which the slp_tsk system call has placed in a wakeup wait state to enter READY state. The return position in the ready queue is the last position of the task queue having the same priority.
Tasks in WAIT-SUSPEND state go to SUSPEND state.
If the specified task has not executed slp_tsk and is not in a wait state, this wakeup request is queued. A queued wakeup request becomes effective when the specified task executes slp_tsk thereafter. Consequently, the specified task is not placed in a wait state by this slp_tsk.

Note: By default, the number of times wakeup requests are queued (wupcnt) is 1. However, this setting can be customized so that they will be queued up to 255 times. (Refer to Section 2.4, "Customizing ROS33".)

Suspend Other Task sus_tsk

Format: `ER sus_tsk( ID tskid );`

Parameter: `ID tskid` Task ID number

Return values:

`E_OK`	Terminated normally
`E_ID`	Illegal ID number (tskid is illegal or cannot be used)
`E_NOEXS`	Specified task does not exist.
`E_OBJ`	Specified task is the issuing task itself or in DORMANT state.
`E_QOVR`	SUSPEND request is issued more than once.

Description: This system call causes the task specified by tskid to enter SUSPEND state. If you specify a task that is already in WAIT state, the task enters WAIT-SUSPEND state.
SUSPEND state is cleared by issuing the rsm_tsk system call.
SUSPEND requests cannot be nested (cannot be preissued a number of times).

Resume Suspended Task rsm_tsk

Format: `ER rsm_tsk( ID tskid );`

Parameter: `ID tskid` Task ID number

Return values:

`E_OK`	Terminated normally
`E_ID`	Illegal ID number (tskid is illegal or cannot be used)
`E_NOEXS`	Specified task does not exist.
`E_OBJ`	Specified task is not in SUSPEND state.

Description: This system call frees the task specified by tskid from SUSPEND state and returns it to the state it was in when sus_tsk was issued. If the task is WAIT-SUSPEND state, it enters WAIT state.
If you specify a task that is neither in WAIT state nor in WAIT-SUSPEND state, error code E_OBJ is returned.

Cancel Wake Up Request can_wup

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

Parameters:

`INT *p_wupcnt`	Pointer to number of times current wakeup request is issued
`ID tskid`	Task ID number

Return values:

`E_OK`	Terminated normally
`E_ID`	Illegal ID number (tskid is illegal or cannot be used)
`E_NOEXS`	Specified task does not exist.
`E_OBJ`	Specified task is in DORMANT state.

Description: This system call clears the wakeup request counter of the task specified by tskid and invalidates the queued task wakeup request. The wakeup request count before being cleared is set in *p_wupcnt. By specifying TSK_SELF (0) for tskid, you can clear the wakeup request for the issuing task itself.

3.4.3 System Calls of Synchronization and Communication Functions

Wait on Semaphore — wai_sem
Poll and Request Semaphore — preq_sem

Format:
```
ER wai_sem( ID semid );
ER preq_sem( ID semid );
```

Parameter: `ID semid` Semaphore ID number

Return values:

E_OK	Terminated normally
E_ID	Illegal ID number (semid is illegal or cannot be used)
E_NOEXS	Specified semaphore does not exist.
E_RLWAI	Wait state is forcibly cleared (rel_wai accepted during wait state).
E_TMOUT	Failure during polling.
E_CTX	Context error (executed from a task-independent portion or when dispatch is disabled)

Description: The wai_sem system call acquires one resource from the semaphore specified by semid.
If a resource exists, that is, the semaphore counter = 1 or greater, the counter is decremented by 1 and the system call is terminated immediately. This means that a resource has been acquired, so that the task continues executing. If no resource exists, i.e., the semaphore counter = 0, the task is removed from the ready queue and placed in a semaphore queue. This task enters a wait state. If the semaphore counter becomes 1 or greater and there is no other task at the top of the queue waiting for the same semaphore, the semaphore counter is decremented and the task is freed from the wait state. The task is placed back in the ready queue at the last position of the task queue having the same priority. If the task has been in WAIT-SUSPEND state, it enters SUSPEND state.
The preq_sem system call is a polling version of wai_sem and does not have a function to enter a wait state. If a resource has been acquired, it functions the same way as wai_sem. If it cannot acquire any resources, it returns error code E_TMOUT.

Note: Although, by default, up to eight semaphores can be used, it can be customized up to 255 semaphores (semaphore ID = 1 to 255). (Refer to Section 2.4, "Customizing ROS33".)
The initial value and maximum value of a semaphore are set to 1 by default. They can be customized up to 255. However, these values must have the same value.

Signal Semaphore — sig_sem

Format:
```
ER sig_sem( ID semid );
```

Parameter: `ID semid` Semaphore ID number

Return values:

E_OK	Terminated normally
E_ID	Illegal ID number (semid is illegal or cannot be used)
E_NOEXS	Specified semaphore does not exist.
E_QOVR	Semaphore count exceeds the maximum value.

Description: This system call returns one resource to the semaphore specified by semid.
If there are no tasks waiting for the semaphore, the number of resources (semaphore counter) is incremented by 1. If there are tasks waiting for the semaphore, the number of resources is left unchanged so as to ensure that the task at the top of the queue will be assigned a resource. The task assigned a resource is removed from the semaphore queue, placed in READY state, and returned to the ready queue. If the task has been in WAIT-SUSPEND state, it enters SUSPEND state.

Note: Although, by default, up to eight semaphores can be used, it can be customized up to 255 semaphores (semaphore ID = 1 to 255). (Refer to Section 2.4, "Customizing ROS33".)
The initial value and maximum value of a semaphore are set to 1 by default. They can be customized up to 255. However, these values must be a same value.

Receive Message from Mailbox — rcv_msg
Poll and Receive Message from Mailbox — prcv_msg

Format:
```
ER rcv_msg( T_MSG **ppk_msg, ID mbxid );
ER prcv_msg( T_MSG **ppk_msg, ID mbxid );
```

Parameters:

`T_MSG **ppk_msg`	Pointer to pointer to message
`ID mbxid`	Mailbox ID number

Return values:

`E_OK`	Terminated normally
`E_ID`	Illegal ID number (mbxid is illegal or cannot be used)
`E_NOEXS`	Specified mailbox does not exist.
`E_RLWAI`	Wait state is forcibly cleared (rel_wai accepted during wait state).
`E_TMOUT`	Failure during polling.
`E_CTX`	Context error (executed from a task-independent portion or when dispatch is disabled)

Description: This system call receives a message from the mailbox specified by mbxid.
If the message box contains messages, the pointer value that indicates the position of the first message is set in **ppk_msg and the system call is terminated immediately. This means that the message has been received, so the task continues executing.
If the message box does not contain a message, the task is removed from the ready queue and placed in the message queue. The task then enters a wait state. If a message is sent along and there is no other task at the top of the queue waiting for the same message, the pointer that indicates the position of the message is set in **ppk_msg and the task is freed from the wait state. The task is placed back in the ready queue at the last position of the task queue having the same priority. If the task has been in WAIT-SUSPEND state, it enters SUSPEND state.
The prcv_msg system call is a polling version of rcv_msg and does not have a function to enter a wait state. If a message is successfully received, it functions the same way as rcv_msg. If it cannot receive a message, it returns error code E_TMOUT.

Note: Although, by default, up to eight mailboxes can be used, it can be customized up to 255 mailboxes (mailbox ID = 1 to 255). (Refer to Section 2.4, "Customizing ROS33".)

Send Message to Mailbox — snd_msg

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

Parameters:

`ID mbxid`	Mailbox ID number
`T_MSG *pk_msg`	Pointer to message

Return values:

`E_OK`	Terminated normally
`E_ID`	Illegal ID number (mbxid is illegal or cannot be used)
`E_NOEXS`	Specified mailbox does not exist.
`E_PAR`	Parameter error (value that cannot be used by pk_msg)

Description: This system call sends a message to the mailbox specified by mbxid.
If there are tasks waiting for the message, the message is sent to the task at the first position. This task is removed from the message queue, becomes READY, and is placed back into the ready queue. If the task has been in WAIT-SUSPEND state, it enters SUSPEND state.
Note that it is the pointer *pk_msg that is registered in the queue, and not the body of the message.

Note: The message must be initialized before it can be used. Initialize pk_msg->pNxt to 0 before you start sending.
Although, by default, up to eight mailboxes can be used, it can be customized up to 255 mailboxes (mailbox ID = 1 to 255). (Refer to Section 2.4, "Customizing ROS33".)

Message structure:

The message structure T_MSG is defined in "itron.h" as follows:

```
typedef struct t_msg {
    struct t_msg*  pNxt;           ... Message header
    VB             msgcont[10];    ... Message body
} T_MSG;
```

A message consists of a header (first 4 bytes) and a message body.
To expand a message body into 10 bytes or more, define as follows:
Example:

```
VB      msg_buf[25];
T_MSG*  pk_msg;
pk_msg = (T_MSG*)msg_buf;
```

Wait for Event Flag — wai_flg
Wait for Event Flag (Polling) — pol_flg

Format:
```
ER wai_flg( UINT *p_flgptn, ID flgid, UINT waiptn, UINT wfmode );
ER pol_flg( UINT *p_flgptn, ID flgid, UINT waiptn, UINT wfmode );
```

Parameters:

`UINT *p_flgptn`	Pointer to flag pattern
`ID flgid`	Event flag ID number
`UINT waiptn`	Flag wait bit pattern
`UINT wfmode`	Flag wait mode and whether or not cleared

Return values:

`E_OK`	Terminated normally
`E_ID`	Illegal ID number (flgid is illegal or cannot be used)
`E_NOEXS`	Specified flag does not exist.
`E_PAR`	Wait pattern (waiptn) is 0 or wfmode specification is illegal.
`E_OBJ`	Object status is invalid. (Multiple tasks waiting for event flag of TA_WSGL attribute)
`E_RLWAI`	Wait state is forcibly cleared (rel_wai accepted during wait state).
`E_TMOUT`	Failure during polling.
`E_CTX`	Context error (executed from a task-independent portion or when dispatch is disabled)

Description:

This system call waits until the event flag specified by flgid is set to a specified state.
Use waitptn and wfmode to set the conditions under which you want to exit a wait state. For wfmode, one of the following four conditions can be set:

1. TWF_ANDW	AND condition Wait until all of the bits that have been set to 1 by waiptn are set.
2. TWF_ANDW | TWF_CLR	AND condition and event flag clear In addition to the TWF_ANDW condition, the event flag is cleared (all bits to 0) when the condition is met.
3. TWF_ORW	OR condition Wait until one of the bits that have been set to 1 by waiptn is set.
4. TWF_ORW | TWF_CLR	OR condition and event flag clear In addition to the TWF_ORW condition, the event flag is cleared (all bits to 0) when the condition is met.

If the condition for exiting a wait state has already been met when this system call is issued, the task continues executing without entering a wait state.
If the condition for exiting a wait state has not been met, the task is removed from the ready queue and placed in a wait queue. This task is kept waiting until the wait clearing condition is met.
When the wait clearing condition is met, the task waiting for the relevant event flag is freed from wait state. The task is placed back in the ready queue at the last position of the task queue that has the same priority. If the task has been in WAIT-SUSPEND state, it enters SUSPEND state.

The event flag, that existed when the wait clearing condition was met, is returned to the pointer *p_flgptn. Even if you specify TWF_CLR, the bit pattern that existed before being cleared when the AND or OR condition was met is returned.

The pol_flg system call is a polling version of wai_flg and does not have a function to enter a wait state. If the wait clearing condition was met, it functions the same way as wai_flg. If the condition was not met, it returns error code E_TMOUT.

Note:

Although, by default, up to eight event flags can be used, it can be customized up to 255 event flags (event flag ID = 1 to 255). (Refer to Section 2.4, "Customizing ROS33".)
The event flags in ROS33 are one byte long (8 bits).
ROS33 does not allow multiple tasks to wait for the same event flag.

Set Event Flag set_flg

Format: `ER set_flg( ID flgid, UINT setptn );`

Parameters:

`ID flgid`	Event flag ID number
`UINT setptn`	Bit pattern to be set

Return values:

`E_OK`	Terminated normally
`E_ID`	Illegal ID number (flgid is illegal or cannot be used)
`E_NOEXS`	Specified flag does not exist.

Description: This system call sets the bits specified by setptn of the event flag. This event flag is specified by flgid. This setting is made by a logical OR, so that the bits set to 1 by setptn are set and those set to 0 do not change their state. If at this time there is a task waiting for the flag, the wait pattern and wait condition are checked. The task is removed from the flag wait queue and returned to the ready queue if the wait condition is met. If any task was previously in WAIT-SUSPEND state, it enters SUSPEND state.

Note: The event flags in ROS33 are one byte long (8 bits).
ROS33 does not allow multiple tasks to wait for the same event flag.

Clear Event Flag clr_flg

Format: `ER clr_flg( ID flgid, UINT clrptn );`

Parameters:

`ID flgid`	Event flag ID number
`UINT clrptn`	Bit pattern to clear

Return values:

`E_OK`	Terminated normally
`E_ID`	Illegal ID number (flgid is illegal or cannot be used)
`E_NOEXS`	Specified flag does not exist.

Description: This system call clears the bits specified by clrptn of the event flag. This event flag is specified by flgid. This clearing is made by a logical AND, so that the bits set to 0 by clrptn are cleared and those set to 1 do not change state. The clr_flg system call does not dispatch the task even if the wait condition is met.

Note: The event flags in ROS33 are one byte long (8 bits).
ROS33 does not allow multiple tasks to wait for the same event flag.

3.4.4 System call for Extension Synchronization and Communication

Send Messages to Message Buffer — snd_mbf
Send Messages to Message Buffer (Polling) — psnd_mbf
Send Messages to Message Buffer (with timeout) — tsnd_mbf

Format:

```
ER snd_mbf( ID mbfid, VP msg, INT msgsz );
ER psnd_mbf( ID mbfid, VP msg, INT msgsz );
ER tsnd_mbf( ID mbfid, VP msg, INT msgsz, TMO tmout );
```

Parameters:

ID	mbfid	Message buffer ID
INT	msgsz	Size of sent message(byte)
VP	msg	Start address of sent message
TMO	tmout	Specification of time out

Return values:

E_OK	Terminated normally
E_ID	Invalid ID number (tskid is illegal or cannot be used)
E_NOEXS	Specified task does not exist.
E_PAR	Parameter (msgsz is less than 0, msgsz is larger than maxmsz specified with cre_mbf(), tmout is less than -2.)
E_RLWAI	Wait state is forcibly cleared (rel_wai accepted during wait state).
E_TMOUT	Failure during polling or excess of timeout.
E_CTX	Context error (snd_mbf is executed from a task-independent portion or when dispatch is disabled)

Description:

This system call sends a message in the msg address to the message buffer specified by mbfid in the specified size (byte) by msgsz . The msg is copied to the message buffer specified by mbfid.
If the empty space is few and msg message cannot be copied to the message buffer, the task is placed in the buffer queue.
If enough space is generated for buffer, the task is freed from the wait state. If there is a message larger than the empty buffer space and there is no other message smaller than the empty buffer space at the top of the waiting queue, the messages behind it are freed from the wait state. If multiple tasks waiting to send message are freed from the wait state at the same time, the task order in the ready queue after exiting the wait state becomes the order of the message queue. If the task has been in the WAIT-SUSPEND state, it enters the SUSPEND state.
The psnd_mbf system call is a polling version of snd_mbf and does not have a function to enter a wait state. If there is enough empty space, it functions in the same way as snd_mbf. If it does not have enough empty space, it returns the error code E_TMOUT.
The tsnd_mbf system call is a system call of snd_mbf with timeout function. Specify the timeout interval in units of 1 ms.psnd_mbf and tsnd_mbf(tmout=TMO_POL) system calls can be executed from a task-independent portion or when dispatch is not permitted.

Receive Messages from Message Buffer rcv_mbf
Receive Messages from Message Buffer (Polling) prcv_mbf
Receive Messages from Message Buffer (with timeout) trcv_mbf

Format:

```
ER rcv_mbf ( VP msg, INT *p_msgsz, ID mbfid );
ER prcv_mbf( VP msg, INT *p_msgsz, ID mbfid );
ER trcv_mbf( VP msg, INT *p_msgsz, ID mbfid, TMO tmout );
```

Parameter:

`VP msg`	Address to store the received message
`INT *p_msgsz`	Size of the received massage (byte)
`ID mbfid`	Message Buffer ID
`TMO tmout`	Specify Timeout

Returned values:

`E_OK`	Terminated normally
`E_ID`	Invalid ID number (tskid is illegal or cannot be used)
`E_NOEXS`	Specified task does not exist
`E_PAR`	Parameter (tmout is less than -2.)
`E_RLWAI`	Wait state is forcibly cleared (rel_wai accepted during wait state).
`E_TMOUT`	Failure during polling or excess of timeout.
`E_CTX`	Context error (snd_mbf is executed from a task-independent portion or when dispatch is disabled)

Description:

This system call receives a message from the message buffer specified by mbfid. The message is copied to the address specified by msg. The received message size is housed in p_msgsz. If the message buffer does not contain a message, the task is removed from the ready queue and placed in the message queue. The task then enters a wait state. If a message is sent to the buffer and there is no other task at the top of the queue waiting for the same message, the task is freed from the wait state. The task is placed back in the ready queue at the last position of the task queue having the same priority.

If the task has been in the WAIT-SUSPEND state, it enters the SUSPEND state.

The prcv_mbf system call is a polling version of rcv_mbf and does not have a function to enter a wait state. If there is enough empty space, it functions in the same way as rcv_mbf. If it does not have enough empty space, it returns the error code E_TMOUT.

The trcv_mbf system call is a system call of rcv_mbf with timeout function. Specify the timeout interval in units of 1 ms.

3.4.5 System Calls of System Management Functions

Get Version Information get_ver

Format: `ER get_ver( T_VER *pk_ver );`

Parameters: `T_VER *pk_ver` Beginning address of packet that returns version information

Return values:

`E_OK`	Terminated normally
`E_PAR`	Parameter error (Packet address for return parameter cannot be used)

Description: This system call returns the OS version of the ITRON specification currently being executed.
The following shows the contents of pk_ver:

`maker = 0x0000;`	Manufacturer's code
`id = 0x0001;`	ROS33 type number
`spver = 0x5302;`	µITRON version number (ver 3.02)
`prver = 0x0000;`	ROS33 version number (will be changed by an update)
`prno[0] = 0x0000;`	Unused
`prno[1] = 0x0000;`	Unused
`prno[2] = 0x0000;`	Unused
`prno[3] = 0x0000;`	Unused
`cpu = 0x0000;`	CPU information
`var = 0x8000;`	Variation (level S)

3.4.6 System Calls of Time Management Functions

When using the system calls below, make sure a timer handler is provided in your user program. (Refer to Section 2.3, "Creating an Application Program".)

Set System Clock — set_tim

Format: `ER set_tim( SYSTIME *pk_tim );`

Parameter: `SYSTIME *pk_tim` Packet address indicating the current time

Return values:
`E_OK` Terminated normally
`E_PAR` Parameter error (pk_tim or the set time is illegal)

Description: This system call sets the system clock to the value specified by system.
The system clock is 48 bits long, and the reference time is 1 ms.

Get System Clock — get_tim

Format: `ER get_tim( SYSTIME *pk_tim );`

Parameter: `SYSTIME *pk_tim` Packet address that returns the current time

Return values:
`E_OK` Terminated normally
`E_PAR` Parameter error (pk_tim is illegal)

Description: This system call returns the current system clock value to pk_tim.

Delay Task — dly_tsk

Format: `ER dly_tsk( DLYTIME dlytim );`

Parameter: `DLYTIME dlytim` Delay time (in ms)

Return values:
`E_OK` Terminated normally
`E_PAR` Parameter error (dlytim < 0)
`E_CTX` Context error (executed from a task-independent portion or when dispatch is disabled)
`E_RLWAI` Wait state is forcibly cleared (rel_wai accepted during wait state).

Description: This system call causes the issuing task itself to temporarily stop executing and enter a wait state. Use dlytim to specify how long you want the task to stop executing. Specify this time in units of 1 ms. If the specified time elapses, the task is returned to the ready queue. If the task has been placed in WAIT-SUSPEND state while waiting for the time to expire, it enters SUSPEND state. You can use rel_wai to forcibly clear the state while waiting for the time to expire.

3.4.7 System Calls of Interrupt Management Functions

Return from Interrupt Handler ret_int

Format: `void ret_int( void );`

Parameter: None

Return value: None

Description: System call terminates the interrupt handler. If the dispatch condition is met by the system call issued in the interrupt handler, the dispatch delays until the ret_int system call terminates the interrupt handler. These dispatch requests are handled together when they return from the interrupt handler by ret_int. Specify the values of register %r0, %r15, %alr %ahr, and %sp determined just before you issue ret_int as the same values just after an interruption occurs. When the interrupt handler starts, the OS does not intervene. Save or restore register in the interrupt handler. (See 2.3 Creating an Application Program.)

Lock CPU loc_cpu

Format: `ER loc_cpu( void );`

Parameter: None

Return values:

`E_OK`	Terminated normally
`E_CTX`	Context error (issued from a task-independent portion)

Description: This system call disables external interrupts and task dispatches.
Once this system call is made, the issuing task itself will never be changed from RUN state to READY state, even if some other task with higher priority becomes READY. The task is also disabled from entering WAIT or DORMANT state. If an external interrupt is requested during this time, the corresponding interrupt handler is initiated only when the task is freed from this disable state.
To reenable interrupt and dispatch, use the unl_cpu system call. The dispatch disable state set by loc_cpu cannot be freed by ena_dsp.
If loc_cpu is issued when the task is disabled for dispatches by dis_dsp, the task is disabled for interrupts as well. In this case, too, use unl_cpu to exit the disabled state.

Note: Changing the IE flag by directly accessing the CPU's PSR is prohibited.

Unlock CPU unl_cpu

Format: `ER unl_cpu( void );`

Parameter: None

Return values:

`E_OK`	Terminated normally
`E_CTX`	Context error (issued from a task-independent portion)

Description: This system call reenables external interrupts and task dispatches. This system call can be used to clear the disabled state set by either loc_cpu or dis_dsp.

Note: Changing the IE flag by directly accessing the CPU's PSR is prohibited.

3.4.8 Implementation-Dependent System Calls

Initialize Interrupt Handler Value — ent_int

Format: `void ent_int( void );`

Parameter: None

Return value: None

Description: This system call increments the variable g_ubIntNestCnt, which is used to examine interrupt nesting, and set the system ready for interrupt. This system call automatically saves and returns the necessary register required internally. Issue this system call when interrupt is not permitted. Issue ent_int( ) before enabling external interrupt(specifying PSR IE bit 1) or before issuing system calls.

Create Task — vcre_tsk

Format: `ER vcre_tsk( ID tskid, FP task, PRI itskpri, UW istkadr );`

Parameters:

`ID tskid`	Task ID number
`FP task`	Task startup address
`PRI itskpri`	Priority at task startup (1 to 8, the smaller the value, the higher the priority)
`UW istkadr`	Initial stack address

Return value: `E_OK` Terminated normally

Description: This system call defines a specific task ID specified in tskid and a task which has a default priority specified in itskpri. It is important for user to maintain a necessary size of stack used by each task. Specify a default address in istkadr.
A defined task becomes the DORMANT state.
Priority can be changed after starting a task. However, priority returns to the value which is set here when a task becomes the DORMANT state.

Note: This system call can be called only from the user's main( ) function.
Use of task ID created by except vcre_tsk( ) is not guaranteed.

System Clock — sys_clk

Format: `void sys_clk( );`

Parameters: None

Return value: None

Description: It is necessary to set a timer handler in the user program and call back sys_clk( ) per 1ms for using the time management system.
This system call automatically saves and returns register required internally.
Specify 1ms for the reference time for SYSTIME and DLYTIME used in the time management system. Call back cycle can be changed by "INT_TIME" definition of "ros33.h". Make sure that it is a multiple of 8. The reference time for the system clock is still 1ms.

Define Message Buffer vcre_mbf

Format: `ER vcre_mbf( ID mbfid, VP msgbuf, INT bufsz, INT maxmsz );`

Parameters:

`ID mbfid`	Message Buffer ID
`VP msgbuf`	Start address of ring buffer containing a message
`INT bufsz`	Size of ring buffer(byte)
`INT maxmsz`	Maximum size of message(byte)

Return value:

`E_OK`	Terminated normally
`E_NOEXS`	Specified mbfid is larger than maximum message buffer number (MSGBUF_NUM).
`E_PAR`	Parameter is invalid (bufsz is not larger than maxmsz+4)

Description: This system call defines a message buffer which has an ID number specified in mbfid. Specifically, it sets a ring buffer which contains a message for a message buffer control block specified in mbfid. It is necessary for bufsz to set more than maxmsz+4.
The sent message is contained in this ring buffer. The size information is contained in the first 4 bytes, and subsequent bytes are used to store message sent by the user.
bufsz=0 message buffer can be created. In this case, communication for sending and receiving are completely synchronized in the message buffer.

Ring buffer condition containing a message:

`size 4bytes`	`message`	`empty...`

Note: This system call can be called only from user's main( ) function.
Message buffer ID created not from vcre_mbf( ) cannot be used.

Change Semaphore Count Value vchg_semcnt

Format: `void vchg_semcnt( ID semid, UB semcnt );`

Parameters:

`semid`	Semaphore ID number
`semcnt`	Semaphore count value (1 - 255)

Return value:

`E_OK`	Terminated normally
`E_ID`	Invalid ID number(semid is invalid or can not be used)
`E_NOEXS`	Specified Semaphore is not found.

Description: This system call is executed between sys_ini( ) and sys_sta( ) in the main function.
It is used to change the semaphore count value to a value other than the value defined in ros33.h SMPH_CNT.
Semaphore count value with no change by this function is still SMPH_CNT.

Time management function check vchk_timer

Format: `int vchk_timer( void );`

Parameters: None

Return value:

`1`	This means a task wait for time.
`0`	This means no task wait for time.

Description: This system call checks if there is a wait for a time-out by the system clock or if there is a task waiting by dly_tsk.

3.4.9 System Calls for Only Interrupt Handler

Interrupt handler for only wup_tsk — iwup_tsk

Format: `ER iwup_tsk( ID tskid );`

Parameters:

`ID tskid`	`Task ID Number`

Return value:

`E_OK`	Terminated normally
`E_ID`	Invalid ID number (tskid is invalid or can not be used)
`E_NOEXS`	No specified task found
`E_OBJ`	Specified task is the issuing task or DORAMNT state
`E_QOVR`	Wake up requests exceed the allowable

Description: This system call is for interrupt handler only. Use this call after %r12 for the parameter and %r10 which stores the returned value.
This system call saves and returns other registers required for the execution.

Interrupt handler for only snd_msg — isnd_msg

Format: `ER isnd_msg( ID mbxid, T_MSG *pk_msg );`

Parameters:

`ID mbxid`	Mailbox ID number
`T_MSG *pk_msg`	Pointer to the message

Return value:

`E_OK`	Terminated normally
`E_NOEXS`	No specified mailbox is found
`E_PAR`	Parameter (The value which can not use pk_msg

Description: This is snd_msg for interrupt handler only. Use this after %r12, %r13 for the parameter and %r10 which stores the returned value.
This system call saves and returns other registers required for the execution.

Interrupt handler for only set_flg — iset_flg

Format: `ER iset_flg( ID flgid, UINT setptn );`

Parameters:

`ID flgid`	Event flag ID number
`UINT setptn`	Bit pattern which is set

Return value:

`E_OK`	Terminated normally
`E_ID`	Invalid ID number (flgid is invalid or can not be used)
`E_NOEXS`	No specified flag found

Description: This is set_flg for interrupt handler only. Use this after %r12, %r13 for the parameter and %r10 which stores the returned value.
This system call saves and returns other registers required for the execution.

Interrupt handler for only sig_sem — isig_sem

Format: `ER isig_sem( ID semid );`

Parameters:

`ID semid`	Semaphore ID number

Return value:

`E_OK`	Terminated normally
`E_ID`	Invalid ID number (semid is invalid or can not be used)
`E_NOEXS`	No specified semaphore found
`E_QOVR`	Count value of Semaphore exceed the maximum.

Description: This is sig_sem for interrupt handler only. Use this after %r12 for the parameter and %r10 which stores the returned.
This system call saves and returns other registers required for the execution..

Interrupt handler for only psnd_mbf — ipsnd_mbf

Format: `ER ipsnd_mbf(ID mbfid, INT msgsz, VP msg);`

Parameters:

`ID mbfid`	Message buffer number
`INT msgsiz`	Size of sent message(byte)
`VP msg`	Start address of sent message

Return value:

E_OK	Terminated normally
E_ID	Invalid ID number (mbfid is invalid or can not be used)
E_NOEXS	No specified message buffer found
E_PAR	Parameter error
E_TMOUT	Polling failed

Description: This is sig_sem for interrupt handler only. Use this after %r12,%r13, %r14 for the parameter and %r10 which stores the returned value.
This system call saves and returns other registers required for the execution .

EPSON International Sales Operations

AMERICA

EPSON ELECTRONICS AMERICA, INC.

- HEADQUARTERS -
1960 E. Grand Avenue
El Segundo, CA 90245, U.S.A.
Phone: +1-310-955-5300 Fax: +1-310-955-5400

- SALES OFFICES -

West
150 River Oaks Parkway
San Jose, CA 95134, U.S.A.
Phone: +1-408-922-0200 Fax: +1-408-922-0238

Central
101 Virginia Street, Suite 290
Crystal Lake, IL 60014, U.S.A.
Phone: +1-815-455-7630 Fax: +1-815-455-7633

Northeast
301 Edgewater Place, Suite 120
Wakefield, MA 01880, U.S.A.
Phone: +1-781-246-3600 Fax: +1-781-246-5443

Southeast
3010 Royal Blvd. South, Suite 170
Alpharetta, GA 30005, U.S.A.
Phone: +1-877-EEA-0020 Fax: +1-770-777-2637

EUROPE

EPSON EUROPE ELECTRONICS GmbH

- HEADQUARTERS -
Riesstrasse 15
80992 Munich, GERMANY
Phone: +49-(0)89-14005-0 Fax: +49-(0)89-14005-110

- GERMANY -
SALES OFFICE
Altstadtstrasse 176
51379 Leverkusen, GERMANY
Phone: +49-(0)2171-5045-0 Fax: +49-(0)2171-5045-10

- UNITED KINGDOM -
UK BRANCH OFFICE
Unit 2.4, Doncastle House, Doncastle Road
Bracknell, Berkshire RG12 8PE, ENGLAND
Phone: +44-(0)1344-381700 Fax: +44-(0)1344-381701

- FRANCE -
FRENCH BRANCH OFFICE
1 Avenue de l' Atlantique, LP 915 Les Conquerants
Z.A. de Courtaboeuf 2, F-91976 Les Ulis Cedex, FRANCE
Phone: +33-(0)1-64862350 Fax: +33-(0)1-64862355

ASIA

- CHINA -
EPSON (CHINA) CO., LTD.
28F, Beijing Silver Tower 2# North RD DongSanHuan
ChaoYang District, Beijing, CHINA
Phone: 64106655 Fax: 64107319

SHANGHAI BRANCH
4F, Bldg., 27, No. 69, Gui Jing Road
Caohejing, Shanghai, CHINA
Phone: 21-6485-5552 Fax: 21-6485-0775

- HONG KONG, CHINA -
EPSON HONG KONG LTD.
20/F., Harbour Centre, 25 Harbour Road
Wanchai, HONG KONG
Phone: +852-2585-4600 Fax: +852-2827-4346
Telex: 65542 EPSCO HX

- TAIWAN -
EPSON TAIWAN TECHNOLOGY & TRADING LTD.
10F, No. 287, Nanking East Road, Sec. 3
Taipei, TAIWAN
Phone: 02-2717-7360 Fax: 02-2712-9164
Telex: 24444 EPSONTB

HSINCHU OFFICE
13F-3, No. 295, Kuang-Fu Road, Sec. 2
HsinChu 300, TAIWAN
Phone: 03-573-9900 Fax: 03-573-9169

- SINGAPORE -
EPSON SINGAPORE PTE., LTD.
No. 1 Temasek Avenue, #36-00
Millenia Tower, SINGAPORE 039192
Phone: +65-337-7911 Fax: +65-334-2716

- KOREA -
SEIKO EPSON CORPORATION KOREA OFFICE
50F, KLI 63 Bldg., 60 Yoido-dong
Youngdeungpo-Ku, Seoul, 150-763, KOREA
Phone: 02-784-6027 Fax: 02-767-3677

- JAPAN -
SEIKO EPSON CORPORATION
ELECTRONIC DEVICES MARKETING DIVISION

Electronic Device Marketing Department
IC Marketing & Engineering Group
421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN
Phone: +81-(0)42-587-5816 Fax: +81-(0)42-587-5624

ED International Marketing Department Europe & U.S.A.
421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN
Phone: +81-(0)42-587-5812 Fax: +81-(0)42-587-5564

ED International Marketing Department Asia
421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN
Phone: +81-(0)42-587-5814 Fax: +81-(0)42-587-5110

In pursuit of **"Saving" Technology**, Epson electronic devices.
Our lineup of semiconductors, liquid crystal displays and quartz devices
assists in creating the products of our customers' dreams.
Epson IS energy savings.

EPSON

SEIKO EPSON CORPORATION
ELECTRONIC DEVICES MARKETING DIVISION

■ EPSON Electronic Devices website
http://www.epson.co.jp/device/

First issue FEBRUARY 1999 Ⓜ, Printed JUNE 2000 in Japan ⓌA