μITRON4.0 Specification Real Time OS

NORTi Version 4 User's Guide (Kernel Edition)

Preface

"NORTi Version 4", a product that is confidently supplied to you by MiSPO Co., Ltd., is the real-time OS based on the μITRON specifications as exhibited by TRON association. This product has implemented all the system calls in the μITRON 4.0 specifications (except the definition of the CPU sample handler). Furthermore, it is compatible with system calls of NORTi3 (μITRON3.0 specification), so that the previous version of software components can be utilized without any modification.

NORTi is a compact and development friendly OS designed exclusively for Embedded Systems. Just similar to compiler library, NORTi OS functions are operational after linking NORTi libraries with user application program.

NORTi includes the TCP/IP protocol stack conforming to "ITRON TCP/IP API specification" and is suitable for operations with Embedded Systems. Using NORTi, the correspondence is very fast for embedded systems development using network connection with indispensable technology.

For your system developments, please use the highly efficient and compact NORTi OS, which comes with all source code as standard attachment without any royalty charges.

About This Documentation

This book (Kernel edition) is a common reference manual for real-time multitasking functions of NORTi Version 4 series. The first half section explains the outline and each system call is explained in second half section. Please refer to the installed document about a report peculiar to a processor. Please refer to the user's guide of Network edition for detailed information about a TCP/IP protocol stack functions.

Reference

The support window at MiSPO Co., Ltd. via E-mail is open at following addresses.

General inquiry: sales@mispo.co.jp   Technical support request: norti@mispo.co.jp

Please enquire at individual manufacturer, when NORTi is introduced as a bundled product with debugger or hardware board etc.

Index

Preface

Index

1. Basic Particulars

2. Introduction

3. Task and Handler Description

4. Function Overview

5. System Call Description

6. Exclusive System Calls

7. List

1. Basic Particulars

1.1 Features

High Speed Response

NORTi is preemptive multitasking RTOS. Scheduling is carried out based on the priority of the events and highest priority task is immediately activated. All kernel source code is fully tested. CPU performance is pulled to the maximum extent. Interrupt of priority higher than kernel level can be processed with interrupt inhibit time conventionally reduced to half. Furthermore, the interrupt routine with priority higher than OS can be carried with unlimited value of interrupt-prohibition time.

Compact Size

Kernel size is effectively optimized since all management block variables (i.e. TCB etc) are inside kernel. All variables are optimized for size by 1 byte margin in order to effectively use precious RAM area.

Kernel Designed with C source code

All major source code of Kernel is described in C programming language and is very easy to understand. It is misunderstanding that OS designed by C code is inferior to OS designed by assembly code. In contrary, high speed can be achieved by the proper management of the internal register switching / restoration and with the allocation management of the unused registers to the compiler. Compatibility with new CPU is the other advantage gained by C language code. Since the source code is common for two or more types of CPUs, it is reliable even after release of new version of CPU.

Conformity to both μITRON4.0 and μITRON3.0 Specifications

μITRON4.0 specifications of TRON association have neglected conformity to 3.0 specifications. However in case of NORTi, not only μITRON4.0 specifications but also interface to μITRON3.0 specifications is mounted. In addition the software programs are designed to maintain compatibility with previous versions.

Full Set of μITRON

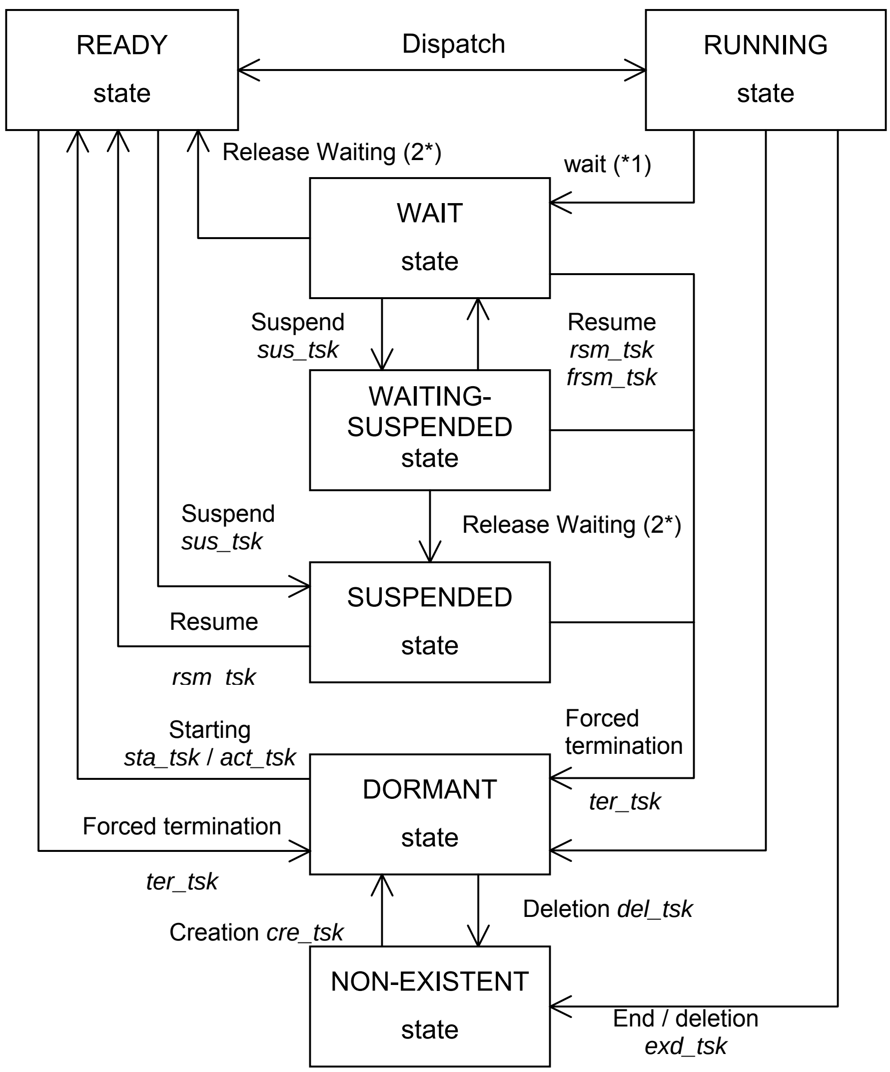
While observing the μITRON specifications, excluding the mounting of troublesome part, among OS which has μITRON API with different architecture as in Japan, the full set of functions as per μITRON 4.0/3.0 are set in NORTi very carefully with additional various synchronous communication methods. (A definition of CPU exception handler is removed)

Corresponds to verities of processors, Compilers and Debuggers

Since NORTi is already corresponded to many 16 & 32-bit processors commonly used in industry, it can be used without any change even if the target system differs. Moreover in order to provide support to wide range of development environment tools, continuous effective correspondence is performed in association with almost all development toolmakers.

1.2 Task States

A program is executed concurrently in units called tasks. A task can take on any of seven states namely NON-EXISTENT, DORMANT, READY, RUN, WAIT, SUSPEND and WAITING-SUSPEND. The following diagram illustrates the state transitions of tasks.



(*1) slp_tsk, tslp_tsk,wai_sem, twai_sem, wai_flg, twai_flg, rcv_mbx, trcv_mbx, rcv_mbf, trcv_mbf, snd_mbf, tsnd_mbf, cal_por, tcal_por, acp_por, tacp_por, get_mpl, tget_mpl, get_mpf, tget_mpf, dly_tsk, snd_dtq, tsnd_dtq, rcv_dtq, trcv_dtq, loc_mtx, tloc_mtx

(*2) rel_wai, wup_tsk, sig_sem, set_flg, del_sem, snd_mbx, snd_mbf, tsnd_mbf, psnd_mbf, rcv_mbf, prcv_mbf, trcv_mbf, del_mbf, cal_por, tcal_por, acp_por, tacp_por, del_por, rpl_rdv, rel_mpl, del_mpl, rel_mpf, del_mpf, snd_dtq, psnd_dtq, tsnd_dtq, del_dtq, unl_mtx, del_mtx, ter_tsk

Ready to Run State (READY)

The task is ready to execute, but is not being executed either because a task with a higher priority or the one with the same priority is being executed.

Run State (RUNNING)

The task in this state is currently executing with the assigned processor. Only one RUN state task exists at one time. For tasks, there is no big difference between the READY state and the RUN state. The READY state task with the highest priority can be also regarded as the RUN state task.

Wait State (WAITING)

The task is blocked from executing by a system call issued by the task itself. For event driven multitasking, once tasks are started, they ought to remain in the WAIT state for most of the time. If not, other tasks cannot execute during the waiting time.
Wait states are classified by the following catagories.

Wakeup wait (slp_tsk, tslp_tsk)
Wait for fix Time (dly_tsk)
Event flag creation wait (wai_flg, twai_flg)
Semaphore acquisition wait (wai_sem, twai_sem)
Waiting for Mutex acquisition (loc_mtx, tloc_mtx)
Waiting while receiving a message at mailbox (rcv_mbx, trcv_mbx)
Waiting while receiving a message at message buffer (rcv_mbf, trcv_mbf)
Waiting while sending a message from message buffer (snd_mbf, tsnd_mbf)
Waiting while sending a data queue (snd_dtq, tsnd_dtq)
Waiting while receiving a data queue (rcv_dtq, trcv_dtq)
Waiting for a rendezvous call (cal_por, tcal_por)
Waiting for a rendezvous acceptance (acp_por, tacp_por)
Waiting for a rendezvous end (cal_por, tcal_por)
Waiting while getting fixed-length memory block (get_mpf, tget_mpf)
Waiting while getting variable-length memory block (get_mpl, tget_mpl)

Suspend State (SUSPENDED)

It is the state where execution is suspended from other tasks. The task while in suspended state is hardly used. As an example, temporary suspension of a task for the purpose of debugging can be considered as one of the application.

Suspended Wait State (WAITING-SUSPENDED)

Although it is divided for the sake of management, WAITING-SUSPENDED state is treated same as the SUSPENDED state. The task goes to WAITING-SUSPENDED state if the task is in WAITING state (instead of READY state) when suspended from the other tasks. It is not necessarily suspended til waiting. If the waiting conditions are fulfilled, the task state will separate only from WAITING and will move to SUSPENDED state.

Dormant State (DORMANT)

In the DORMANT state, tasks do not start or have already been terminated. A task, which is executing can be put in the DORMANT state by a system call issued by the same task itself. In addition it can be forced into the DORMANT state by a system call issued by another task.

Non-Existent State (NON-EXISTENT)

NON-EXISTENT is the state where the task is not generated or has been deleted.

Task Switching Instances

Since NORTi is preemptive-multitasking OS, task with higher priority can interrupt the execution of the running task.
There are following four instances when the task switching occurs.

(1) During execution of a task if the task of the higher priority is started, or if the system call is issued so as to cancel the WAIT state of the higher priority.

(2) From a non-task context (Interrupt handler / Interrupt service routine / Timer event handler), if a task with priority higher than the running task is started, or if a system call is published to cancel the wait state of higher priority task.

(3) If the wait-state of the higher priority task is cancelled by the timeout event.

(4) If the task under execution went into wait-state by itself, or if the priority is lowered, or if is terminated.

In other words, all system calls does not necessarily cause task switching. Even if the task of lower priority is started or is released from wait-state, task switching does not occur. Task switching operation will be waiting until the operated task is higher priority as in (4) above.
Although the case of similar priority is same as the case of low priority, the task switching between same priorities can occur by using rot_rdq and chg_pri system calls, where task under execution moves to the end of execution queue.

Differences from NORTi3

Following names were changed,

RUN → RUNNING, WAIT → WAITING

1.3 Terminology

Object and ID

Generally objects are the targets of system call operations. The numbers, which are used to identify and distinguish objects, are called IDs. These IDs are user specified numbers. A part which is internal to Kernel and which cannot be directly specified by user is called an Object number.

Objects with ID number include tasks, semaphores, event flags, a mailboxes, message buffers, rendezvous ports, fixed-length / variable-length memory pools, data queue, mutex, cyclic handlers, alarm handlers and interrupt service routines. The objects identified by object numbers are interrupt-handlers, rendezvous ports and statically generated interrupt service routines.

Context

The entire execution environment of the task at a given point of time is called the “context” of that task. In concrete terms, this can be understood as registers of the CPU. Context is a generic name of things saved or restored when tasks are switched.

Under multitasking, using DSP and floating-point arithmetic requires the registers to switch their contexts. If NORTi does not support this switching operation, a floating-point unit needs to be exclusively controlled.

Task Independent Context

Interrupt handler, timer handler sections altogether are task independent context or non-Task context. There are three types of timer handler namely cyclic handler, alarm handler and overrun handler. (In case of μITRON3.0 specifications, a task independent section, time event handler and timer handler altogether are called non-task context.)

Since each of the non-task context handlers is not a task, the system calls referring to the self-task cannot be issued.

In addition, by μITRON specification Task independent system call is distinguished by first character as ‘i’. In case of NORTi, since the context is automatically distinguished inside a system call, system call with ‘i’ (starting character) is treated same as the system call without ‘i’ by kernel.h

Dispatch

Selection and change of an execution task is called Dispatch. Some system calls generates dispatch and some do not generate dispatch. Task will not change if the priority of the task from which dispatch generating system call is issued, is lower than the priority of the current RUNNING task. In addition, if the system call, which generates the dispatch, is issued from the non-task context, dispatch is carried out collectively after returning to task context. This is called the delayed context.

Synchronization / Communication Functions

The synchronization function is used for enabling synchronization and communication between tasks. The communication function is used for sending and receiving data between tasks. Since the synchronization function is also used for communication, both the functions are described collectively.

Programs can be carefully designed by using global variables and made to wait for sending and receiving data between tasks without using synchronization / communication function. However, using OS functions is easier, safer and elegant.

There are 7 types of synchronization and communication mechanisms i.e. semaphores, event flags, mailboxes, message buffers, rendezvous ports, data queue and mutex.

Queue

Tasks are queued (put in a waiting line) in the order of arrival when multiple tasks make their requests to the same object. Queues are created when waiting for semaphores, waiting for event flags, waiting for message from mailboxes, waiting for transmission / reception of messages from message buffers, ports waiting for rendezvous call / reception, waiting for memory block acquisition from fixed-length / variable-length memory pools, waiting for reception / transmission of data queues and waiting for mutex acquisition.

Tasks are basically queued on the FIFO (First in First Out) basis. However in case of semaphore, mailbox, message buffer (reception side), fixed-length / variable-length memory pool and mutex, it is possible to set the queue in the order of task or message priority.

Queuing

Queuing means a reservation of a request from a task without considering the state erroneous where the request cannot be received by other task.

Requests for waking up tasks and messages at the mailbox / message buffer and data queue are queued. Requests for waking up tasks are implemented by counting requests. Messages for mailboxes are queued by linear linked lists with pointers. Messages for message buffers and data queuing are queued by a ring buffer.

In case of event flag and task exception, instead of queuing, the event by OR operation and suspension of cause of exception is carried out. In this case, only the existence of the event is recorded, the counting is not recorded.

Polling and Timeout

In system calls where waiting may occur, the function of polling without waiting and the timeout function are provided. In case of Polling, if waiting occurs, it is regarded as an error.

Parameter and Return-Parameter

As per μITRON specification, data transferred from the user is called parameter, and data returned from system calls is called return parameter. In this book it is considered as general arguments of C language function or procedure.

Since the return value of a system call is basically an error code, for the returned value other than the error code, the data location of the return parameter is specified as an argument.

System Call and Service Call

The interface (API) between the kernel and the application software is called a service call. The service call of the kernel specifically is called a system call.

Exclusive Control

Multitasking may allow multiple tasks to access an object that is not to be accessed simultaneously. However, there are many objects that cannot be used concurrently. Example: non-reentrant functions and commonly shared data. Exclusive control manages these object resources in such a way that they cannot be used concurrently. Semaphores or mutex are generally used for the exclusive control management.

However exclusive control is unnecessary if tasks priorities are the same or if the competing tasks are not switched while accessing shared resources. Unifying priorities effectively prevents the use of exclusive control. In some case, it is better to raise the priority of competing section temporarily. For example semaphores have a problem with priority reversal i.e. tasks with high priority must wait for semaphores return of low priority task. The so-called momentary dispatch-disabled / prohibited interrupt disabled state makes exclusive control easy if the interrupt is short. In case of mutex, there is an option of raising the priority whenever required. However, if the section, which should carry out exclusive control, is short then it is easy to carry out exclusive control by temporary ban on dispatch or temporary ban on interruption.

Idle Task

An idle task executes when no other tasks are running. Although there is an idle task implemented in the kernel, if a user creates a task as an infinite loop operation and with lowest priority, it will serve as an idle task.

Though an idle task does not do anything, it plays an important role. In event driven multitasking system, if an execution order does not turn to an idle task, then it indicates that either some task is consuming CPU power wastefully or CPU performance is not up to system requirement.

Static Error and Dynamic Error

System calls return two types of errors i.e. static errors and dynamic errors.

Static errors are generally the abnormalities of the parameters regardless of the system state. For example an ID number is out of its valid range. Static errors can be rectified using debugging.

A dynamic error is an error that occurs depending on system states or timings. For example wait-state cancellation of a task even before the task gets into the WAIT state. Some programs are created to positively use dynamic errors such as polling failure.

In order to achieve high-speed execution, NORTi provides a library that does not check static parameter errors.

Context Error

There are some system calls, which cannot be issued from a non-task context (timer handler or interrupt handler). Violation of this rule returns a context error from system calls. Since this is a static error, libraries in which static parameters are not checked do not detect context errors.

Static API and Dynamic API

In μITRON specification, system call described by uppercase letters is the static API but is not necessarily directly supported by OS. In case of static API structure, the management block of TCB etc. is secured during compilation and initialization at the time of system starting is premised. That is, code generated attached with static API is necessary before compilation and for this purpose Configurator was introduced in μITRON4.0 specification.

Since the basic concept used in NORTi is generation of dynamic objects, at the time of initialization, NORTi configurator changes the code of static API described by configuration file to usual dynamic API.

1.4 Common Conventions

System call name

The ITRON system calls are named in the basic format of xxx_yyy, where xxx is an abbreviation for the method of the operation, and yyy is an abbreviation for the object subjected to operation. A system call derived from xxx_yyy becomes zxxx_yyy by adding a one-character prefix. The first character of a system call to be polled is ‘p’. The first character of a system call with timeout is ‘t’ and that of an original system call is ‘v’.

Data type name

As per ITRON Data type naming conventions, only uppercase letters are used. The data types of pointers are named as ~ P. The data types of structures are basically named as T_ ~.

Argument name

Following convention is used for naming input arguments to system calls.

p_~	Pointer to the location of data storage
pk_~	The pointer to a packet (structure object)
ppk_~	The pointer to the place which stores the pointer to a packet (structure object)
~id	ID
~no	Number
~atr	Attribute
~cd	Code
~sz	Size (in Bytes)
~cnt	Number
~ptn	Bit Pattern
i~	Initial value

Handling zeros and negative numbers

In the input and output of system calls, zeroes often have a special meaning. For example, the task ID of a task itself is specified as zero. A task itself / local task mean the task issuing this system call. IDs and priorities begin with '1' to allow zero to have a special meaning. Moreover, by ITRON specification, negative value is taken as “System” value. Error code of the system call is negative.

In addition, as per μITRON3.0 specifications, system objects negative ID numbers (-1) ~ (-4) are reserved. However this condition is removed in μITRON3.0 specifications and is not used by NORTi either.

1.5 Data Types (for 32-bit CPU)

In ITRON, system calls are declared by using redefined types as given below. INT, UINT are 32-bit data types.

General purpose data type

typedef signed char B;	8-bit signed integer
typedef unsigned char UB;	8-bit unsigned integer
typedef short H;	16-bit signed integer
typedef unsigned short UH;	16-bit unsigned integer
typedef long W;	32-bit signed integer
typedef unsigned long UW;	32-bit unsigned integer
typedef char VB;	Type undefined data (8-bit size)
typedef int VH;	Type undefined data (16-bit size)
typedef long VW;	Type undefined data (32-bit size)
typedef void *VP;	Pointer to type undefined data
typedef void (*FP)();	Start address of the program in general

ITRON dependent data types

typedef int INT;	Signed integer
typedef unsigned int UINT;	Unsigned integer
typedef int BOOL;	Boolean value (FALSE(0) or TRUE(1))
typedef INT FN;	Function code
typedef int ID;	Object ID number
typedef int RDVNO;	Rendezvous number
typedef unsigned int ATR;	Object attribute
typedef int ER;	Error code
typedef int PRI;	Task priority
typedef long TMO;	Timeout
typedef int ER_ID;	Error code or object ID number
typedef long DLYTIME;	Delay time
typedef unsigned int STAT;	State of an Object
typedef unsigned int MODE;	Operation mode of a Service call
typedef unsigned int ER_UINT;	Error code or an unsigned integer
typedef unsigned int TEXPTN;	Task Exception pattern
typedef unsigned int FLGPTN;	Event flag bit pattern
typedef unsigned int RDVPTN;	Rendezvous pattern
typedef unsigned int INHNO;	Interrupt handler number
typedef unsigned int INTNO;	Interrupt number
typedef VP VP_INT;	Task parameter and extended information
typedef unsigned long SIZE;	Size of a memory domain

** Previous to NORTi Kernel 4.05.00, MODE was incorrectly mounted to INT.

** Although ER_BOOL is defined in ITRON specification, it is not used in NORTi.

Time related data types

```
typedef struct t_systim         System clock and system time
{       H utime;                Upper 16bit
        UW ltime;               Lower 32bit
}SYSTIM;

typedef long RELTIM;            Relative time
typedef long OVRTIM;            Overrun time
```

Differences from NORTi3

Structure objects CYCTIME and ALMTIME were unified to integer type RELTIM.

Renaming was done for SYSTIME → SYSTIM, RNO → RDVNO and HNO → INHN.

BOOL_ID was discontinued.

VP_INT, ER_ID, ER_UINT, SIZE, MODE, STAT, FLGPTN, RDVPTN, TEXPTN, OVRTIM were newly added.

Particularly, be careful about not to use data types SIZE and MODE as macro definitions in user program.

1.6 Data Types (for 16-bit CPU)

INT and UINT data types are 16 bits size. Since *int* and *short* are same, H and UH are considered as *int* data type instead of *short*.

General purpose data types

Typedef signed char B;	8-bit signed integer
Typedef unsigned char UB;	8-bit unsigned integer
Typedef int H;	16-bit signed integer
Typedef unsigned int UH;	16-bit unsigned integer
Typedef long W;	32-bit signed integer
Typedef unsigned long UW;	32-bit unsigned integer
Typedef char VB;	Type undefined data (8-bit size)
Typedef int VH;	Type undefined data (16-bit size)
Typedef long VW;	Type undefined data (32-bit size)
Typedef void *VP;	Pointer to type undefined data
Typedef void (*FP)();	Start address of the program in general

ITRON-dependent data types

Typedef int INT;	Signed integer
Typedef unsigned int UINT;	Unsigned integer
Typedef int BOOL;	Boolean value (FALSE(0) or TRUE(1))
Typedef int ID;	Object ID number
Typedef int RDVNO;	Rendezvous number
Typedef unsigned int ATR;	Object attribute
Typedef int ER;	Error code
Typedef int PRI;	Task priority
Typedef long TMO;	Timeout
Typedef long DLYTIME;	Error code or object ID number
Typedef int ER_ID;	Delay time
Typedef unsigned int STAT;	State of an Object
Typedef unsigned int MODE;	Operation mode of a Service call
Typedef unsigned int ER_UINT;	Error code or an unsigned integer
Typedef unsigned int TEXPTN;	Task Exception pattern
Typedef unsigned int FLGPTN;	Event flag bit pattern
Typedef unsigned int RDVPTN;	Rendezvous pattern
Typedef unsigned int INHNO;	Interrupt handler number
Typedef unsigned int INTNO;	Interrupt number
Typedef VP VP_INT;	Task parameter and extended information
Typedef unsigned long SIZE;	Size of a memory domain

** Previous to NORTi Kernel 4.05.00, MODE was incorrectly mounted to INT.

** Although ER_BOOL is defined in ITRON specification, it is not used in NORTi.

Time related data types

typedef struct t_ ystem	System clock and system time
{ H utime;	Upper 16bit
UW ltime;	Lower 32bit
}SYSTIM;	
typedef long RELTIM;	Relative time
typedef long OVRTIM;	Overrun time

Differences from NORTi3

Same as described earlier in case of 32-bit CPU.

(Blank space)

2. Introduction

2.1 Installation

NORTi installation standard folder composition is explained in the following text.

/NORTi/INC	INCLUDE files
/NORTi/SRC	source files
/NORTi/SMP/XXX/BBB	Sample
/NORTi/LIB/XXX/YYY	Library
/NORTi/DOC	Document

XXX is the processor series name (Example: SH, H8S, H83 etc.), BBB is the evaluation board name (Example: MS7709A etc.) and YYY is the name of corresponded compiler in short (Example: SHC, GHS, GCC etc.).

The portion described as xxx in the file name is processor/device dependent. Extensions are typical examples and actually depend on the compiler. Refer to the supplementary documentation or README text for up-to-date information about the folder contents. Please do not inter-mix the files with same name. The same name may exist for the files of different versions, files of different processor and the files of NORTi3 Standard / Extended / Network.

Include files

Following header files are stored in INC folder.

itron.h	ITRON standard header file
kernel.h	Kernel standard header definitions
nosys4.h	System internal definition header file
nocfg4.h	Configuration header file
n4rxxx.h	CPU dependent definition header file
no4hook.h	HOOK routine definition header file
norti3.h	Kernel standard header file for NORTi3 compatibility
nosys3.h	System internal definition header file for NORTi3 compatibility
nocfg3.h	Configuration header file for NORTi3 compatibility
n3rxxx.h	CPU dependent definition header file for NORTi3 compatibility
no3hook.h	HOOK routine definition header file for NORTi3 compatibility
nosio.h	Serial I/O function header file
non????.h	Network header file (Refer to network user’s guide)

#include "kernel.h" in all source files using NORTi. It describes all definitions and declarations necessary for using NORTi functions such as data types, common constants and function prototypes. Since itron.h is included in kernel.h, it is not necessary to #include itron.h in user’s source files.

“nocfg4.h” defines the default constants for the configuration of the maximum number of tasks and the variable itself used in the kernel. When configurator is not used, #include "nocfg4.h" in only one file of the user programs.

When using configurator constant other than the default, define it before #include. When the configurator is used, it is included in the kernel_cfg.c created by the configurator. Therefore it is unnecessary to #include directly from the user program.
nosys4.h describes all internal definitions of the kernel. It is included in nocfg4.h and usually it is unnecessary to carry out #include directly from user's programs. The part, which changes with the corresponding processor, is defined in n4rxxx.h. It is included from nosys4.h and is unnecessary to carry out #include directly from user's programs.

Library

The Kernel library module file along with the makefile to generate it is stored in LIB folder.

n4exxx.lib	Kernel library
n4exxx.mak	The makefile which generates above library
n4fxxx.lib	Kernel library without parameter check
n4fxxx.mak	The makefile which generates above library
n4nxxx.???,n4dxxx.???	Network library (Refer to network user’s guide)

Depending on the compiler, library module may have extension other than lib.
Library command file has dependency with compiler.

A library without parameter check is a library in which static eror check of a parameter is omitted for the sake of processing speed improvement. If an error code is not set to SYSER variable unique to NORTi, then it is okay to switch to library without parameter check.

Source files

The SRC directory contains all source files of the kernel.

n4cxxx.asm	A CPU interface module
noknl4.c	NORTi Kernel source
non????.c	Network stack source files (Refer to Network user’s guide)

Depending on the compiler / assembler, the assembler source file may have extension other than asm.

Sample

The cyclic timers interrupt handler and interrupt management function modules, which are dependent on the hardware, should be fundamentally created by the user. For designing these modules, please refer to following source / header files provided as a sample.

n4ixxxx.c	Interrupt management function / cyclic timer interrupt handler source
nosxxxx.c	Serial I/O driver source (optional)
nosxxxx.h	Serial I/O driver header (optional)

Apart from this, header files defining the corresponding processor's built-in I/O, start-up routine samples, main source sample, and make files are also included.

2.2 Kernel configuration

As opposed to other operating systems based on the μITRON 4.0 specification, NORTi does not adopt troublesome configuration procedures. All that you have to do is to #define all the required configurations and #include "nocfg4.h" in one of the source files of the user programs usually the file that includes the "main" function.

When using the software components such as network, the ID number used by the user program and the ID number used in the software component should not mismatch. In such cases, it is possible to automatically allocate the ID numbers by using the configurator. Please refer to the configurator manual that is attached. The kernel configuration for system without the configurator is explained in the following text.

Default configuration values

If the following standard configuration values are sufficient, then only necessary thing to do is #include "nocfg4.h".

Task ID	8
Timer handler number upper limit	1
Each of the Other ID's	8
Task Priority upper limit	8
Interrupt handler stack size	4 times the size of T_CTX (*1)
Timer handler stack size	4 times the size of T_CTX
System memory size	0(using stack memory)
Memory Size of memory pool	0(using stack memory)
Stack memory size	0(using default stack) (*2)

(*1) T_CTX is defined in n4rxxx.h and the size is the same as that of the sum total of the total CPU registers size except a stack pointer (SP).
(*2) A default stack usually points at the start address of the stack section specified by the linker to the address set up by SP at the time of reset.

Customization of configuration

Upper / lower limits of the IDs and numbers are as follows:

Task ID / Timer event handler ID	1 to 253(*3)
Other object IDs	1 to 999(*4)
Task priority	1 to 31

(*3) This ID is managed by 1 byte and 255 and 254 are used for special processing inside.
(*4) In addition, although ID is unrestricted as a matter of fact to a memory bound because of management by int, the guarantee is taken as to 3 digit figures.

For the upper limit of the task priority, specify smallest possible value. With higher number of maximum priority, the time to choose the highest priority task is also higher. Besides, the internal data size, which controls waiting queues in the priority order, increases one byte per priority.

For definitions other than task priority definitions, there is no speed overhead due to excessive upper limit. However since one pointer is internally defined for each ID, systems of a smaller RAM capacity should adopt minimum definition values as illustrated below.

```
#define TSKID_MAX 16        Task ID upper limit
#define SEMID_MAX 4         Semaphore ID upper limit
#define FLGID_MAX 5         Event flag ID upper limit
#define MBXID_MAX 3         MailBox ID upper limit
#define MBFID_MAX 2         Messenger buffer ID upper limit
#define PORID_MAX 2         Rendezvous ID upper limit
#define MPLID_MAX 3         Variable size memory pool ID upper limit
#define MPFID_MAX 3         Fixed size memory pool ID upper limit
#define DTQID_MAX 1         Data queue ID upper limit
#define MTXID_MAX 1         Mutex ID upper limit
#define ISRID_MAX 1         Interrupt service routine ID upper limit
#define SVCFN_MAX 1         Extended service call routine ID upper limit
#define CYCNO_MAX 2         Cyclic handler ID upper limit
#define ALMNO_MAX 2         Alarm handler ID upper limit
#define TPRI_MAX 4          Task priority maximum
#include "nocfg4.h"
```

Timer queue size

In order to implement timeout or timer handlers, three kinds of timer queues are available. If RAM is sufficient, change the size of queues to 256 in order to substantially improve the processing speed of the timeout function or time management function. Please set numeric values as a power of 2 (1, 2, 4, 8, 16, 32, 64, 128, 256). See the example below.

```
#define TMRQSZ 256          Timer queue size of the task
#define CYCQSZ 128          Timer queue size of a cyclic handler
#define ALMQSZ 64           Timer queue size of a alarm handler
              :
#include "nocfg4.h"
```

Interrupt handler stack size

The stack size of the interrupt handler is defined as 4 times the context (T_CTX) size by default. When the RAM capacity is insufficient, carefully reduce this value.

At the time of system initialization, the stack of the interrupt handler is dynamically reserved from the "stack memory." All the interrupt handlers share this stack area. If there are multiple interrupts, consider that the stack size of the interrupt handlers needs an additional area to be reserved for nesting.

```
#define ISTKSZ 400          Stack size for interrupt handler
              :
#include "nocfg4.h"
```

Timer event handler stack size

The stack size of the timer event handler (cyclic handler and alarm handler) is by default defined as 4 times the context (T_CTX). If the RAM capacity is insufficient, carefully reduce the value.

At the time of system initialization, the stack of the timer handler is dynamically reserved for the "stack memory." All the timer handlers share the stack area. The time handler is not put in a nested state. An example is shown below.

```
#define TSTKSZ 300          Stack size for timer event handler
        :
#include "nocfg4.h"
```

System memory and management block sizes

The management blocks for a task, a semaphore, an event flag, etc. are all dynamically allocated from the "system memory" provided by the OS. Based on the following table, total a required block sizes, and define a numeric value more than the total value in size SYSMSZ of the system memory. The table shows the minimum size of each management block.

[1]	[2]	
40	40	x Number of tasks
12	12	x Number of semaphores
16	12	x Number of mutex
12	8	x Number of event flags
12	12	x Number of mailbox
24	24	x Number of message buffers
28	28	x Number of data queue
12	12	x Number of rendezvous ports
20	16	x Number of Variable length memory pools
20	18	x Number of Fixed length memory pools
32	28	x Number of Cyclic handlers
12	12	x Number of alarm handlers
8	8	x Number of extended service calls
20	18	x Number of interrupt service routines
16	14	x Number of Task exception handler routines

[1] In the case of pointer 32-bit, INT type integer 32-bit (SH, 68K, V800, PowerPC, ARM, MIPS etc.)
[2] In the case of pointer 32-bit, INT type integer 16-bit (H8S, H8/300H, 8086, etc.)

The size of (1 byte x task priority upper limit TPRI_MAX) is added to the management block of an object created by specifying the task priority wait. If the sum total size is not multiple of int size, it is realigned. When the object creation information exists in RAM instead of ROM, the object creation information is copied to the system memory.

The amount of system memory used is decided by number of objects created simultaneously. Although 8 is specified as the upper limit of the object number, if it does not generate simultaneously, it is not necessary to secure 8 objects. Defining 0 in SYSMSZ makes the system memory allocated from the "stack memory." Hence in most of the cases SYSMSZ definition is unnecessary.

Following is the example of a definition.

```
#define SYSMSZ 2352          System memory size
          :
#include "nocfg4.h"
```

Memory size of a memory-pool

The memory blocks of the fixed-length / variable-length memory pools and ring buffer area of message buffer are allocated from the "memory for the memory pool" provided by the OS. Please define the size that is essential for application. Since with the default value of 0, the memory pool is allocated from the "stack memory", in most of the cases it is not necessary to define MPLSZ.

```
#define MPLMSZ 2048          Memory size of a memory pool
          :
#include "nocfg4.h"
```

Size of a stack memory

The task for stack when stack domain is not specified by cre_tsk / interrupt handler stack / timer handler stacks are allocated from the "stack memory" provided by the OS.

Define a total value of the stack size required for an application task plus a stack size required for the interrupt handler/timer handler. The system memory when STKMSZ=0 and the memory pool memory when MPLMSZ=0 are also allocated from this stack memory. The default value is 0. In this case, the stack memory of OS is the standard stack area decided by the initial stack pointer value setup by linker and the startup routine.
In addition, even when STKMSZ is other than 0, in order to allocate stack to main function, timer handler uses default stack area of the processing system.

```
#define STKMSZ 2048          Stack memory size
          :
#include "nocfg4.h"
```

About dynamic memory management

With repeated generation and deletion memory fragmentation of system memory, memory for memory pool and the stack memory may not be avoided. As an example, although sum total size is sufficient, size of a successive empty domain is small, and it may stop allocating big size memory. Moreover, the processing time of the dynamic memory management is dependent on the status of the memory assignment at that time. It is not possible to reduce maximum value of the processing time.

Therefore it is recommended to create all objects collectively at the time of system start, and to avoid repeated creation and deletion during user program.

Interrupt-inhibit level of a kernel

In a critical partition inside the kernel, interrupts are temporarily prohibited. You can select the interrupt prohibition level of the kernel in the processors having level interrupt function. However, a system call cannot be issued with an interrupt routine having a higher priority than the kernel.

Note that, when the priority of interrupt handlers is kept high, lowering only the interrupt level of the kernel will cause overrun.

```
          :
#define KNL_LEVEL 6       Kernel interrupt-inhibit level
          :
#include "nocfg4.h"
```

ID Definition

The μITRON 4.0 specification requires the ID's to be predetermined. You can #include the header files that #define all the IDs from the source files of the user program.

(Example-1)

```
- kernel_id.h -              - Each Source -
#define ID_MainTsk 1         #include "kernel.h"
#define ID_KeyTsk 2          #include "kernel_id.h"
#define ID_ConSem 1
#define ID_KeyFlg 1                      :
#define ID_ErrMbf 1
          :
```

In case when configurator is used, static API of a configuration file generates kernel_id.h automatically.

If ID is defined as a global variable, all files need not be re-compiled when ID value is changed.

(Example-2)

```
- xxx_id.c -                 - Each Source -
#include "kernel.h"          #include "kernel.h"
ID ID_MainTsk = 1;           extern ID ID_MainTsk;
ID ID_KeyTsk = 2;            extern ID ID_KeyTsk;
ID ID_ConSem = 1;                        :
ID ID_KeyFlg = 1;
ID ID_ErrMbf = 1;
          :
```

Automatic assignment of ID

You may receive unused ID number as a return value when you create objects by acre_xxx system call. For this you do not have to define ID numbers beforehand. It is advisable to refer to ID numbers as a global variable, as shown in (Example 2).

Empty identification number is checked in descending order. This improvement easily avoids a conflict between automatically assigned ID numbers and ID numbers defined in ascending order.

2.3 Example of creation of user program

Following is an easy example using two tasks. Task2 cancels the waiting of task1.

```
#include "kernel.h"
#include "nocfg4.h"

TASK task1(void)                /* Task1 */
{
        FLGPTN ptn;

        for(;;)
        {       tslp_tsk(100/MSEC)
                wai_sem(1);
                wai_sem(1);
                wai_flg(1, 0x01, TWF_ORW,&ptn);
        }
}

TASK task2(void)                /* Task2 */
{
        for (;;)
        {       wup_tsk(1);
                sig_sem(1);
                set_flg(1, 0x0001);
        }
}

const T_CTSK ctsk1 = {TA_HLNG, NULL, task1, 1, 512, NULL};
const T_CTSK ctsk2 = {TA_HLNG, NULL, task2, 2, 512, NULL};
const T_CSEM csem1 = {TA_TFIFO, 0, 1};
const T_CFLG cflg1 = {TA_CLR, 0};

void main(void)                 /* main function */
{
        sysini();               /* System initialization */
        cre_tsk(1, &ctsk1);     /* Create task1 */
        cre_tsk(2, &ctsk2);     /* Create task2 */
        cre_sem(1, &csem1);     /* Create semaphore */
        cre_flg(1, &cflg1);     /* Create event flag1 */
        sta_tsk(1 ,0);          /* Start task1 */
        sta_tsk(2, 0);          /* Start task2 */
        intsta();               /* Start cyclic timer interrupt */
        syssta();               /* Start System */
}
```

Example of compilation

A general example of compiling / linking sample.c in the previous page is given below. Vecxxx.asm and init.c describes the interrupt vector definition and the startup routine. File name of the startup routine changes depending on the compiler or may be included in the standard library of C. n4ixxx.c and n4exxx.lib are a cyclic timer interrupt handler description file and a kernel library respectively. standard.lib indicates standard library of C and the file name may change as per the corresponding compiler.

```
>asm vecxxx.asm
>cc init.c
>cc sample.c
>cc n4ixxx.c
>link vecxxx.obj init.obj sample.obj n4ixxx.obj n4exxx.lib standard.lib
```

Above example shows that user need not understand any special procedure to create multi-tasking programs.

3. Task and Handler Description

The software, which constitutes a system, can be divided into OS program and user program. Generally the task and task exception handler are classified into the user program and the handler is classified into the OS program.

This chapter explains the tasks, which the user must describe, and also explains the clear format for describing the handler.

3.1 Task description

Task description method

Tasks are described in the same way as other C functions except for the following two points, which have to be kept in mind.

- The function type must be TASK, and
- An argument is referred to as an int type or void.

Example of task description

Terminating task type

Although ext_tsk() can be omitted, it is recommended to describe this function in order to maintain the compatibility with NORTi3.

```
TASK task1(int stacd)
{
        :
        :
        ext_tsk();
}
```

Repeating task type

```
TASK task1(int stacd)
{
        for (;;)
        {
                :
                :
        }
}
```

Interrupt mask state

After start the task is in interrupt unmasked state.

Task Exception handler routine

Task exception handler routine can be defined for each task. Task exception handler routine is defined as follows.

```
void texrtn(TEXPTN texptn, VP_INT exinf)
{
        :
        :
}
```

TEXPTN is defined in itron.h as a task exception handler type.

3.2 Interrupt service routine and interrupt handler description

Overview

In the ITRON specification, when an interrupt occurs, system passes the control from an interrupt vector to interrupt handler directly created by user. The user defined interrupt service routine is called after carrying out the process within the kernel.

In the interrupt handler, the storing and restoring of registers (ent_int and ret_int in case of NORTi) need to be described by user. On the other hand, in case of interrupt service routine, since the interrupt handler section inside OS is processed initially, user need not describe the storing / restoring of registers i.e. it can be considered as an ordinary C function. This structure of interrupt service routine is introduced from μITRON4.0 specification.

Since the interrupt handler and interrupt service routine are executed in the interrupt state, only minimal processes should be carried out. After this, a task waiting for an interrupt is woken up and practical interrupt handling is carried out. As a matter of fact, waiting system calls are not allowed in interrupt handlers. Moreover system calls requiring dynamic memory management (creation / deletion of object and variable length memory pool etc.) cannot be issued, either.

Interrupt service routine definition method

Interrupt service routine (ISR) can be described as a general C function as shown below. There is no use restriction of auto variables etc. except performing the same consideration as an ordinary interrupt routine.

```
void isr(VP_INT exinf)
{
        :
        :
}
```

exinf is an extended information specified at the time of ISR creation.

Interrupt mask state

In case of CPU which has only 2 states of interrupt enable / disable, ISR once started is in interrupt prohibited state. In case of CPU having level triggered interrupts, at the startup time of ISR, interrupt level is as per the actual hardware. When the higher priority interrupts are generated, multiplexing of interrupts occur.

Interrupt handler definition method

Interrupt handlers are described in the same way as ordinary interrupt routines except for the following two points.

- The function type must be INTHDR, and

- The function must begin with ent_int and must end with ret_int system calls. (the interrupt handler of priority higher than kernel interrupt prohibition level is removed)

Sample description of interrupt handler

```
INTHDR inthdr1(void)
{
        ent_int();
        :
        :
        ret_int();
}
```

ent_int system call

In order to describe interrupt handler entirly by C, ent_int system call is used at the entry of the interrupt handler and is unique to NORTi.

In ent_int, all registers are saved and a stack pointer is also switched over to the exclusive stack area for interrupt handlers. Thus, it is not necessary to add the amount of area used by interrupt handlers to each task stack.

For processors with many registers, all registers are not saved in ent_int. Only the registers, which the compilers use without saving, are saved. The other registers are saved only when it is decided that a dispatch occurs in the ret_int system call at the end of interrupt. This shortens the processing time of an interrupt handler when there is no dispatch or nested interrupts.

Unnecessary instructions before ent_int

Instructions that destroys registers or changes stack pointer must not be generated before the ent_int system call. As the first measure, please enable optimization option to compile interrupt handler. However, note that optimization may not be effective when compiled with debugging options.

Unnecessary instructions generated at the start of functions may vary depending on the contents of the interrupt handlers, the version of the compiler or the compilation conditions. Be sure to output assembly listing files to confirm that no unnecessary instructions are generated. In some case, RISC processors cannot save registers with just ent_int and the Interrupt function is used in this. In this case it is usual to issue register save instructions before ent_int.

Prohibition of auto variables

When auto variables are defined at the start of interrupt handlers, stack pointers shift from ent_int() hypothetical values. You may define static variables or define auto variables in other functions that are called by interrupt handlers. However, if it is clear that there are no auto variables on the stack but only register variables, they can be used as auto variables.

If interrupt handler functions carry out complex processes then an unexpected instructions may be generated before ent_int. In such cases, you may call the function from the interrupt handler and carry out the actual process there.

Suppression of inline expansion

If you are calling more functions from interrupt handler, the inline expansion of these functions may occur inside an interrupt handler due to compiler optimization. In such cases, please compile a program by providing an option that prohibits in-lining.

Description by partial assembly code

When unnecessary instructions before ent_int cannot be suppressed by any means, you may use interrupt service routine or you may describe only the entry and the exit of interrupt handlers in assembly language and call main C function from there. (Refer to applicable supplementary guides for how to develop assemblers).

When in-line assemblers are available, you can cancel unnecessary commands. For example, the generated ‘push’ command can be cancelled by using ‘pop’ command of an inline assembler etc.

Interrupt mask state

When the CPU has only two states of interrupt (i.e. disable or enable), activated interrupt handlers are in the interrupt-disabled state. If you use multiplexed interrupts, you can mask handled interrupt requests by operating the interrupt controller, and then you can enable interrupts by changing the CPU interrupt mask directly.

When the CPU has level interrupt function, the level of the after returning from ent_int() is the same as the hardware. Multiplexing of interrupts happens if interrupt with a higher priority occur.

3.3 Timer event handler description

Overview

In the μITRON 4.0 specification, there are three types of time event handlers i.e. a cyclic handler that is repeatedly executed, an alarm handler that is executed only once and an over run handler, which executes when a specific task exceeds the specified time.

Timer handlers are executed as task independent sections with higher priority than tasks. Therefore accurate time management is possible by using timer handler. Also, management blocks and stacks require less memory than tasks. However, waiting system calls cannot be issued in timer handlers.

Timer event handler definition method

Please perform the description of the cyclic handler and alarm handler similar to the ordinary interrupt routine. Please describe a timer event handler as the following C function.
'exinf' is the extended information that is specified in timer event handler creation.

```
Void tmrhdr(VP_INT exinf)
{
        :
        :
}
```

Consider the description of the overrun handler in the same manner as the ordinary interrupt routine. Please describe an Overrun handler as the following C function.

```
Void ovrhdr(ID tskid, VP_INT exinf)
{
        :
        :
}
```

'tskid' is the task ID of the task which had used up wait-time, and 'exinf' is the extended information specified in the creation of the task.

Interrupt mask state

The system is put in dispatch-prohibited state and interrupts are in the enabled state until the processing of time handlers is completed. If it is interrupt prohibited within timer handlers, please carry out return it after it is back to the interrupt-enabled state.

Additional note

Since the priority of timer handlers is next to that of interrupt handlers, please minimize the processing of timer handlers and enable the compiler optimization. Unlike an interrupt handler, auto variables can be used without limitation.

3.4 Initialization handler

The ITRON specification does not describe about the system initialization method / processing because of its dependency on the processing system. Thus, the contents of this section are unique to NORTi.

Start-up routine

In some other μITRON specified OS, a dedicated start-up routine is provided and the initialization necessary for multi tasking is carried out. After this, there is a way, which starts the main function as a task.

On the other hand, NORTi does not provide any special start-up routine. All the functions till the main function are executed in the same way as an ordinary program.

main function

In NORTi, main function is used as the multitasking initialization handler. In the main function, system initialization (sysini), I/O initialization, one or more task creation (cre_tsk) and one or more task start (sta_tsk) if necessary, the creation of objects (cre_xxx) such as semaphore, an event flags, starting of cyclic timer interrupt start (intsta) and system start (syssta) are performed. When a configurator is used, configurator in a kernel_cfg.c file creates the main function.

System initialization

At the start of the main function, execute the sysini function to initialize the kernel. From sysini, an intini function is called to initialize the interrupt controller interface depending on the hardware. The standard intini function is included in n4ixxx.c. However, if it is not suitable to the user system, please create it separately.

I/O initialization

When an I/O is to be initialized before multi-task operation, use the main function to initialize it. In case the configurator is used, user function that is registered as ATT_INI static function is called.

Object creation

Creation of objects such as task, semaphore, or event flag can be done not only from the main function but also from within a task.

Dynamic memory management is a result of repeated object creation or deletion, and it is inferior to real-time property. As far as possible, create an object in the main function only once and minimize the subsequent object creation.
When using configurator, object creation as registered by CRE_xxx static API is performed.

Task start

You can start all the tasks to be started in the main function. You can start only one task (that is, main task), and then the remaining tasks can be started from within that task. The task to be started should be created beforehand.

In case of configurator, task starting specifies TA_ACT as the task attribute of CRE_TSK static API.

Cyclic timer interrupt start

Use an intsta function to start cyclic timer interrupt by default.

The modules related to model-dependent cyclic timer interrupt and interrupt management are not included in the library. Compile an accessory n4ixxx.c and link it. If the attached n4ixxx.c does not match, the user should create n4ixxx.c.

When configurator is used, cyclic timer is treated as software part. Please refer to the configurator manual for more details such as start timing etc.

System start

A multi-task operation finally starts when you execute syssta function. The syssta function makes an infinite loop internally and does not return to the main function. (This section is NORTi's default idle task.)

However, if an error occurs in cre_tsk or sta_tsk before executing the syssta function, control returns to the main function without starting the multi-task operation.

Example description of initialization handler

Following is the example of description when not using the configurator.

```
#include "kernel.h"

/*Configuration */

#define TSKID_MAX       2       /* Task ID maximum */
#define SEMID_MAX       1       /* Semaphore ID maximum */
#define FLGID_MAX       1       /* Event flag ID maximum */
#define TPRI_MAX        4       /* Task priority maximum */
#define TMRQSZ          256     /* Task queue size for timer */
#define ISTKSZ          256     /* Interrupt handler stack size */
#define TSTKSZ          256     /* Timer event handler stack size */
#define SYSMSZ          256     /* System memory size */
#define KNL_LEVEL       5       /* Kernel interrupt prohibition level */
#include "nocfg4.h"

/* ID definitions */

#define ID_MainTsk      1
#define ID_KeyTsk       2
#define ID_ComSem       1
#define ID_KeyFlg       1

/*Object creation information*/

extern TASK MainTsk(void);
extern TASK KeyTsk(void);

const T_CTSK ctsk1 = {TA_HLNG, NULL, task1, 1, 512, NULL};
const T_CTSK ctsk2 = {TA_HLNG, NULL, task2, 2, 512, NULL};
const T_CSEM csem1 = {TA_TFIFO, 0, 1};
const T_CFLG cflg1 = {TA_CLR, 0};

/* main (initialization handler) */

void main(void)
{
        sysini();                       /* System initialization */
        cre_tsk(ID_MainTsk,&ctsk1);     /* Task1 creation */
        cre_tsk(ID_KeyTsk,&ctsk2);      /* Task2 creation */
        cre_sem(ID_ConSem,&csem1); /* Semaphore creation */
        cre_flg(ID_KeyFlg,&cflg1);      /* Event flag creation */
        sta_tsk(ID_MainTsk,0);          /* Start main task */
        intsta();                       /* Start periodic timer interrupt */
        syssta();                       /* Start multitasking */
}
```

4. Function Overview

4.1 Task management functions

Overview

Executing the cre_tsk system call creates tasks. Tasks are started by sta_tsk or act_tsk. When act_tsk is used, if the specified task is already in the ready state, the start request is queued. Executing ext_tsk or ter_tsk terminates tasks. Ext_tsk terminates the task itself and ter_tsk terminates other tasks. When the start request terminates the queuing task, it restarts instantly. Can_act is used to cancel the queuing of start request. By using disable dispatch dis_dsp and enable dispatch ena_dsp, tasks are switched only once after several system calls are issued.
By chg_pri changing priority and rot_rdq rotating ready queue, you can control the order in which tasks are executed. In addition, the following system calls are classified into task management functions. Rel_wai forces other waiting tasks to be released. Get_tid gets the ID of a task itself. Ref_tsk references a task's status.

Differences with NORTi3

- The task start request (act_tsk), command which cancels the start request (can_act), and command that refers to a task state (ref_tst) were added.
- The function in which the stack domain is securable in user area was added.
- The option was added in which task can be executed after creation.
- The concept of the present priority was introduced.
- get_pri which refers to the present priority was added.
- It is possible to setup task name.
- The task can be terminated now after return from the task main function.
- The functional classification is changed for dis_dsp, ena_dsp, rot_rdq, get_tid and rel_wai.
- vcre_tsk name is changed to acre_tsk.
- vsta_tsk is removed. Instead, please use sta_tsk.

Task management block

Tasks are controlled on the basis of the information in data tables that are called the task control block (TCB).

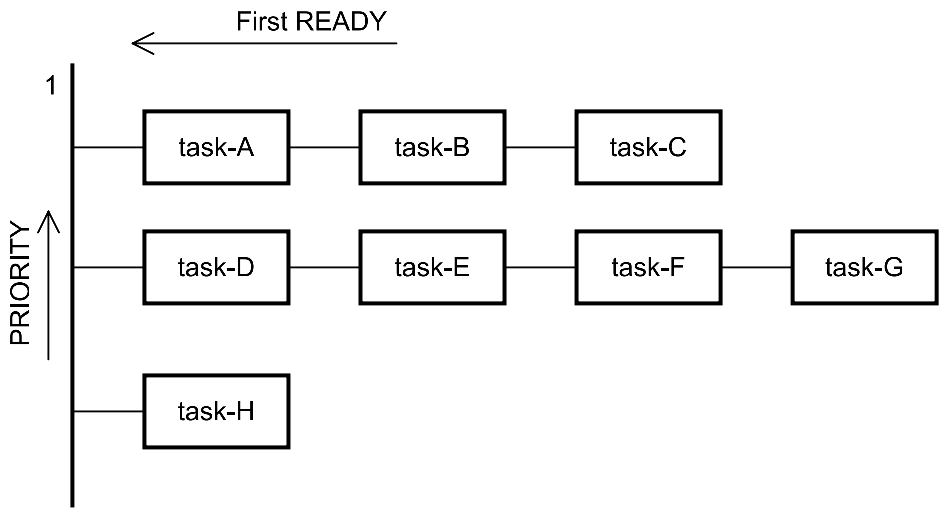
The μITRON specification does not provide a way for users to access TCB and other control blocks directly. Though, in NORTi you can access TCB directly by #including "nosys4.h". The structures of the TCB and others are subject to change by upgraded versions.

Scheduling and ready queue

Scheduling means changing the order of task execution. In ITRON, scheduling is executed based on the priority.

The data structure, which controls the order of execution, is called the ready queue. Tasks are linked to the ready queue in the order of priority. If their priorities are same tasks are linked in the order of FIFO. The READY task with the highest priority is a task in the RUN state (task A in the following chart).

When this task enters the WAIT, SUSPEND or DORMANT state, it is released from the ready queue and the task with the second priority (task B in the following chart) enters the RUNNING state.



Queues for waiting objects with task priority are implemented in the same way as in the ready queue.

4.2 Task dependent synchronization functions

Overview

sus_tsk, rsm_tsk, frsm_tsk, slp_tsk, tslp_tsk, wup_tsk, can_wup, rel_wai and dly_tsk system calls are classified into task-dependent synchronization functions.

Differences with NORTi3

- dly_tsk is classified into the task dependent synchronization function.
- can_wup returns the number of wakeup requests in the queue.
- When dispatch is allowed, a self-task can be specified by sus_tsk.
- A self-task can be specified by wup_tsk.
- rel_wai was classified into the task dependent synchronous function.

Waiting and releasing

Tasks transfer themselves to the WAIT state with the slp_tsk and tslp_tsk system calls. Tslp_tsk can specify a time-out. In other words, it can be used as simple time waiting. But basically dly_tsk must be used for simple time waiting. Tslp_tsk returns E_TMOUT time after the specified time lapses, dly_tsk returns E_OK. Tslp_tsk returns E_OK in the case the wup_tsk is carried out.
As wup_tsk is a queuing function, if wup_tsk is called before calling tslp_tsk, then it returns E_OK in the value, without entering the WAITING state. Tasks, which are put in the WAIT state by slp_tsk or tslp_tsk, can be released (or woken up) by another system call, wup_tsk.

In addition to slp_tsk and tslp_tsk, other system calls like wai_flg, wai_sem and rcv_msg can transfer tasks to the WAIT state. As opposed to the task in these waiting state, issuing rel_wai instead of wup_tsk, forcibly releases the wait.

Suspend and resume

Sus_tsk is the system call, which interrupts the task execution and moves the task state to compulsory wait state i.e. SUSPENDED state.

The task in the suspended state can be resumed by rsm_tsk or frsm_tsk system calls. The queing treatment is the difference between rsm_tsk and frsm_tsk. In case of frsm_tsk system call, all queings are cancelled and task execution is resumed forcibally. However in case of rsm_tsk, queing is decremented by 1.

Suspended waiting

If a sus_tsk system call is issued while task is in waiting state, it will shift to the double waiting state WAITING-SUSPENDED.

In the state of WAITING-SUSPENDED, similar to WAITING state, resource assignment is performed when the turn comes. The task shifts to SUSPENDED state from WAITING-SUSPENDED state after the resource assignment. Since there are no special measures carried out, please be careful with the task of a WAITING-SUSPENDED state about resource allocation delay etc.

4.3 Task exception handling functions

Overview

The task exception handling function is for interrupting the execution of specified task and to perform the task-exception handler routine. A task exception handler routine is executed in the context of the interrupted task. When the specified task is under waiting state i.e. WAITING etc., task exception handler is not executed and it will kept waiting until the task is in READY state. If the task is in READY state, instead of task main part the exception handling routine is performed previously. Execution to task main part will be continued after return from exception handler routine. Each task can register own exception handler routine.

To support task exception-handling function, the system calls to define task exception handling routine (def_tex), call to request task exception (ras_tex), call which prohibits exception handling (dis_tex), call which checks for the prohibition state (sns_tex) and the system call which refers to the exception handling state.

Differences from NORTi3

This function is newly introduced in μITRON4.0

Start and end of exception handling routine

To start a task exception handler routine, ras_tex is called with the exception factor input showing the type of exception handling. The exception handler routine will actually start when an exception handling is enabled by ena_tex system call with a non-zero exception factor and when a specified task is in RUNNING state. Exception factor is cleared to 0 and exception handling is made to prohibition state after actual start of exception-handler routine. The processing which was being performed before starting an exception-handling routine is continued after return from an exception-handler routine.

In case a large address jump is carried using longjmp instead of return from the exception handler routine, it will continue in the exception handling state and does not return to the exception-handling permission state. Moreover the information before starting the exception-handling routine is lost. For example, when WAITING is carried out by rcv_mbf, the information from the received message is lost. When using longjmp, please terminate the task.

Exception factor

When the time of the exception factor is non-zero, it is considered as exception handler demand. If there is an exception demand in an exception handling prohibition state, an exception demand will be suspended until the exception handling is enabled again. The exception factor is defined by TEXPTN type variable. If the same exception is demanded multiple times, a task exception handler routine cannot recognize the number of times the demand had occurred.

4.4 Synchronization / communication function (Semaphore)

Overview

Semaphores are used for the exclusive control of resources. When several tasks moving asynchronously hold resources that cannot be used at the same time (this might include functions, data, input and output), semaphores have to exclusively control the acquisition and return of resources. Semaphores are set up for resources that should be controlled exclusively.

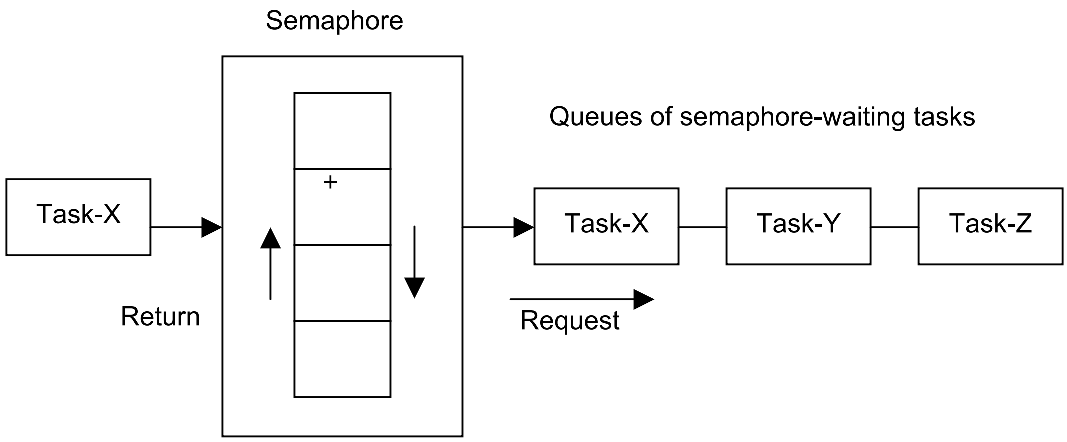
For creating semaphores, the cre_sem and acre_sem system calls are provided. In contrast with the sig_sem system call for returning resources, the wai_sem system call waits for the acquisition of resources, the pol_sem system call executing polling without waiting, and the twai_sem system call waits with a time-out. Besides these, the ref_sem system call references the conditions of a semaphore.

Differences from NORTi3

- Name of system call preq_sem was changed to pol_sem.
- Extended information was deleted from the creation parameter information.
- The extended information was deleted from the information referred by ref_sem.
- When there is no waiting task for the reference information by ref_sem, TSK_NONE is returned instead of FALSE.
- Name of system call vcre_sem was changed to acre_sem.

Semaphore waiting queue

More than a single task can wait for the same semaphore. When FIFO is specified in the creation of semaphores, waiting semaphores are queued in the order that they are requested in. When a task priority is specified in the creation of semaphores, waiting semaphores are queued in the order of the priorities i.e. the first task that issued a request comes before other tasks with the same priority.



Semaphore count value

When sig_sem is performed and there is a task, which is waiting for the semaphore, the task at the top of the queue is changed into a READY state. In case there is no waiting task, the count value of semaphore is incremented by 1.

When wai_sem is performed and the count value of semaphore is 1 or more, then the count value is decremented by 1 while task continues the execution. When the count value is 0, the task goes to WAITING state.

Since the semaphore count values 0 and 1 are enough for general usage, it is recommended to set semaphore maximum = 1.

4.5 Synchronization / communication function (Event flag)

Overview

An event flag is used when you want to inform an opposing task only whether events exist or not.

Event flags are created and deleted with the cre_flg, acre_flg and del_flg system calls. Contrary to the set_flg system call for setting up event flags, the wai_flg system call waits for the existence of event flags, the pol_flg system call executing polling without waiting, and the twai_flg system call waits with a time-out. Besides these, the clr_flg system call clears an event flag and ref_flg references the conditions of an event flag.

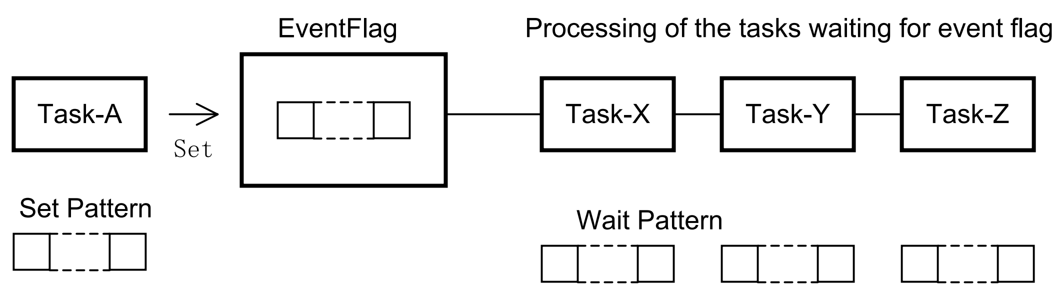
Differences from NORTi3

- Besides the assignment method in the waiting mode of wai_flg, clear specification of an event flag can also assign the generation information.
- Now the task priority level option can be use for the event flag waiting for multiple tasks.
- Extended information was deleted from the generation information.
- Extended information was removed from the information referred to by ref_flg.
- When there is no waiting task, the information referred to by ref_flg returns TSK_NONE instead of FALSE.
- System call name vcre_flg was changed to acre_flg.

Event flag waiting queue

Two or more tasks can wait for the same event flag simultaneously. If the waiting conditions of these tasks are same, then the waiting can be cancelled at once by setting set_flg to 1. However, when clear specification is carried out, the waiting of the task that is previously connected in queue is not cancelled.

However, when the waiting of multiple tasks is cancelled simultaneously, since the system processing time is not minimized, it is recommended for not to use waiting for multiple tasks whenever possible.



Waiting mode

Wait conditions can be specified by bit patterns AND and OR, as multi-bit flag groups are used in an event flag. In waiting AND, the waiting condition waits for all bits specified by a parameter to be set up on event flag. In waiting OR mode, the waiting condition waits for either of the specified bits to be set up on event flag.

Clear order

In the wai_flg, pol_flg and twai_flg system calls, when an event flag has been created, it can be automatically cleared according to the parameter specifications.

When the clear specification is given during creation, it is cleared as usual. Clear is carried out for all bits.

4.6 Synchronization / communication function (Data Queue)

Overview

Data queue is the mailbox implemented using the ring buffer. In order to use the buffer, waiting may occur during transmission as well.

The creation / deletion of data queue are carried out by cre_dtq, acre_dtq and del_dtq. In addition there are system calls to transmit data (snd_dtq), to transmit data by polling way (psnd_dtq), to transmit data with timeout (tsnd_dtq), to wait and receive new message (rcv_dtq), system call to poll and receive new message without waiting (prcv_dtq) and a system call to receive message with timeout (trcv_dtq). Moreover there is fsnd_dtq system call that transmits data forcibily. In addition, ref_dtq system call is available which refers to the data queue state.

Differences from NORTi3

This function is introduced from μITRON4.0

Queuing

Data queue is made up of sending queue, receiving queue and ring buffer. If the buffer is full while sending data, the corresponding task is connected to the send-waiting queue until the data is removed from the buffer. If the buffer is empty while receiving, the receiver task is connected to the receiving queue until the data is transmitted.

The ring buffer size can be set to 0. In this case, the send tasks and receive tasks wait for each other and can be synchronized.

The transmitting queue can specify the task priority or FIFO. The receiving queue is usually formed in the order of arrival.

Data order

Data cannot be assigned priority. However, by using fsnd_dtq, data can be received prior to data sent by snd_dtq. When it is sent by fsnd_dtq, if the buffer is full, the data in the beginning of the buffer is erased and this data is stored there.

4.7 Synchronization / communication function (Mail box)

Overview

Mailboxes are used to send and receive a comparatively large amount of data among tasks. Only the pointer to a data packet, which is called a message, is actually sent and the contents of the message are not copied. For this reason, data can be delivered at high speeds, not depending on the message size. Moreover, a link list of the transmission message from the user area is created. At the time of message transmission there is no waiting for link-list management. Queuing in the mailbox is the processing of message and processing of the task waiting for the reception.

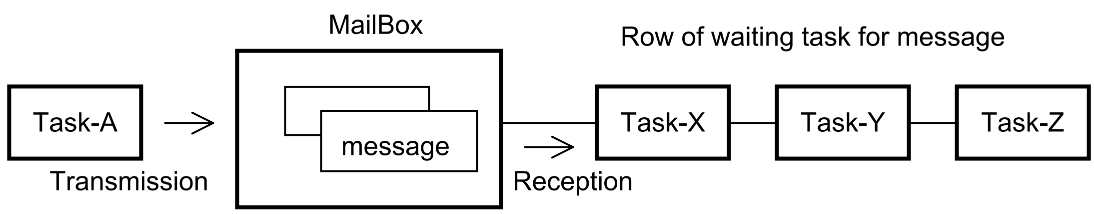
The cre_mbx, acre_mbx and del_mbx system calls are used for creation and dletion of mailboxes. In addition there is system call to transmit message (snd_mbx), call to wait and receive message (rcv_mbx), call to poll and receive the new message (prcv_mbx), call to receive the new message with timeout (trcv_mbx), and a system call to refer to the state of the mailbox (ref_mbx).

Differences from NORTi3

- Extended information was deleted from the mailbox creation information.
- Extended information was deleted from the information referred to by ref_mbx.
- System call name vcre_mbx is changed to acre_mbx.
- System call name xxx_msg was changed to xxx_mbx.

Message queuing

Multiple tasks can wait for the same mailbox. When a FIFO mode is specified at the time of mailbox creation, the queuing is built to serve on the first come first out basis. When the priority mode is specified at the time of mailbox creation, queuing is built as per the task priority (FIFO order among the task with same priority level).



Though both task waiting messages and queued messages are in the chart, they do not exist at the same time.

Message queue

Messages can be sent at any time irrespective of the existence of the receiving task. The top part of the message packet is used as the pointer indicating the next message. Thus, data area on the ROM cannot be used as a message packet.

At the time of mailbox creation, when the FIFO is specified as the queuing method, a message queue is built in the order of arrival.

At the time of mailbox creation, when the priority is specified as the queuing method, a message queue is built in the order of priority. (When the priority is same, queue is built in the order of arrival.) Therefore, the required memory size will increase if the level of priorities is more. The memory size can be known by the TSZ_MPRIHD macro definition.

mprihdsz = TSZ_MPRIHD(8);

Message packet domain

It is not possible to know for certain when a message has been collected in the receiving task. Consequently, it is dangerous to take message packets to auto variables. Besides, even if the message range is defined statically, it is troublesome to check whether it is empty or not and to use it again. When the queued messages are resent, the system operation cannot be guaranteed. Therefore, the memory block acquired from memory pool is ordinarily used for message packets.

The mailbox does not know the message packet size. In other words, the message length is not restricted. However, when it is combined with the fixed-length memory pool, the message packet size is fixed naturally.

4.8 Extended synchronization / communication function (Mutex)

Overview

A mutex is used for an exclusive control of a shared resource such as Semaphore. A difference from semaphore is that mutex supports the mechanism that avoids the task priority inversions, has the ability to lock and automatically unlock the resources. In contrary, semaphore has a resource counter when associated with two or more resources and has the ability to unlock the tasks other than the locked tasks.

Creation and deletion of mutex can be performed using cre_mtx, acre_mtx or del_mtx system calls. In addition there are system calls such as unl_mtx to release the resource, loc_mtx to wait and acquire the resources, ploc_mtx to acquire resource by polling without waiting, tloc_mtx call to acquire by timeout without waiting and ref_mtx call to refer to the state of the mutex.

Differences from NORTi3

This function is introduced from μITRON4.0

Priority inversion

When a low priority task locks the resource, a task with a high priority tends to use the already locaked resources and it may go to WAITING state. At this time, if the task of the priority in between goes to the RUNNING state, then this task indirectly preempts the execution of the high priority task. This is called priority inversion. If a priority inversion happens, operation of the system designed based on the scheduling of priority cannot be guarantied.

In mutex, in order to avoid the priority inversion, the priority inheritance protocol and maximum priority task are supported.

In the priority inheritance protocol, the priority of the locked task is temporarily made the same as the highest priority task among the task waiting for lock release. By this way the intervention of the task with the middle priority is avoided. System is heavily loaded in order to change priority dynamically. Since priority inversion happens when doing changes, cautions are required especially when a task under lock is waiting for another mutex.

In the priority ceiling protocol, the priority of the locked task is changed to the previously decided priority independent of the existence of the waiting task. Although the system is not heavily loaded as compared to the priority inheritance protocol, the priority inversion occurs even when there is no waiting task.

After lock release, the temporarily changed priority will return to the base priority.

4.9 Extended synchronization / communication function (Message buffer)

Overview

A message buffer is an object used for communicating small size messages. The difference from the mailbox is that transmission and reception is performed after the contents of the message are copied to an internal ring buffer. In addition, since the interrupt is prohibited during message copy, please be careful when transfer big size data. With big size data transfer, the interrupt prohibition time will be prolonged.

Message buffers are created and deleted with the cre_mbf, acre_mbf and del_mbf system calls. The snd_mbf system call for sending messages, the psnd_mbf, which returns immediately without waiting in case there is no space in the buffer, the tsnd_mbf which waits with time out when there is no space in the buffer. The rcv_mbf system call waits for the receipt of messages, the prcv_mbf system call executes polling without waiting, and the trcv_mbf system call waits with a time-out. Besides these, the ref_mbf system call references the conditions of message buffers.

Differences from NORTi3

It has become possible to specify the waiting priority even for tasks waiting to send message.

System call name vcre_tsk was change to acre_tsk.

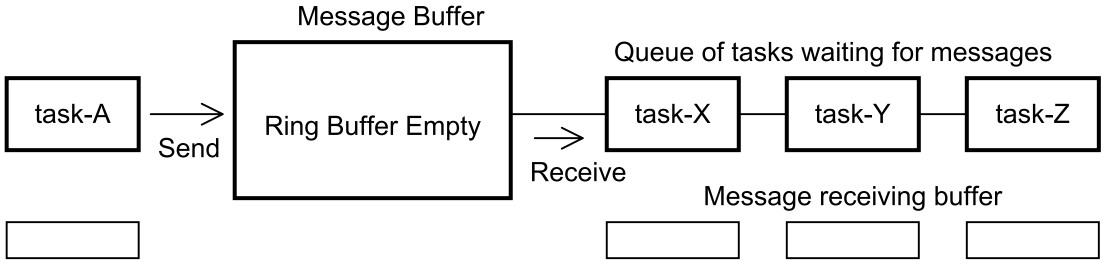
Message queue

Message data is copied into a ring buffer inside the message buffer. Similar to mailbox, it is not necessary to acquire the message packet domain ROM memory pool. Moreover, the message header section used by the OS is also not necessary.

Any message size is acceptable as long as it does not exceed the maximum length specified at message buffer creation in the receiving side. It is necessary to provide a buffer that can receive a message of the maximum length. Only FIFO controls the message line, which has been queued. There is no function to attach priority to the message.

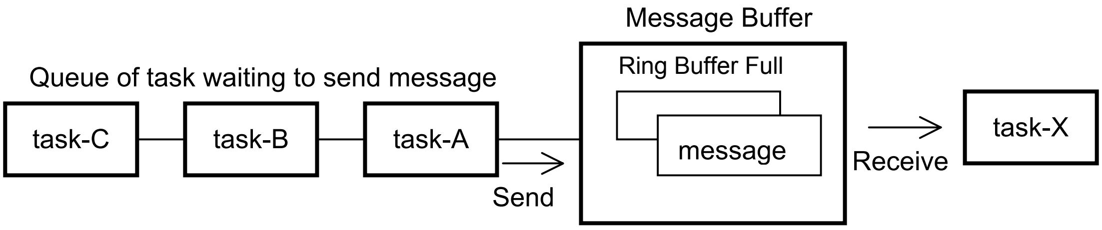
Message reception waiting queue

More than single tasks can wait in the same message buffer. When FIFO is specified in the creation of message buffer, waiting messages are queued in the order of requests received. When a priority is specified in the creation of a message buffer, waiting messages is queued with task priorities (That is, if tasks have the same priority, they are queued in the order that the request is received).



Message transmission waiting queue

Message buffers differ from mailboxes in the point that if there is no space in the ring buffer, the task in the send side also enters the WAITING state. When more than one task wait for sending messages, if FIFO is specified during message buffer creation, these tasks create wait queuing in the order request for sending messages. When priority is specified during message buffer creation, the queue is formed according to the task priority order.



Ring buffer section

A 2-byte header indicating a message size is added to a ring buffer and message data is copied to buffer. Therefore it is not possible to use whole ring buffer domain only for data storage. A ring buffer size, which can store msgcnt messages of msgsz byte size each, can be obtained by TSZ_MBF macro definition. However this is valid only when msgsz is not 1.

TSZ_MBF(msgcnt, msgsz)

When msgsz is one, that is when the message buffer is created with message having maximum length of 1 byte, the addition of the header, which indicates message size, is abbreviated. Because of this function, the entire area of the ring buffer is effectively used for data in the sending and receiving of 1 byte messages.

Ring buffer of size 0

A message buffer can also be generated with ring buffer size=0. In this case the transmitting message is directly copied to the buffer prepared by the receiving side task. For this reason the transmiting task will be waiting until the receiving task is ready to copy message. By this way, a message buffer can realize the synchronous communication similar to rendezvous function.

4.10 Extended synchronization/communication function (rendezvous port)

Overview

Rendezvous port is useful to establish a synchronization and communication between tasks. It also supports mutual data transfer. As by meaning of rendezvous itself, it is mutual waiting between two tasks. Compared to rendezvous functions, other synchronization / communication functions can be treated as single sided waiting and communication functions.

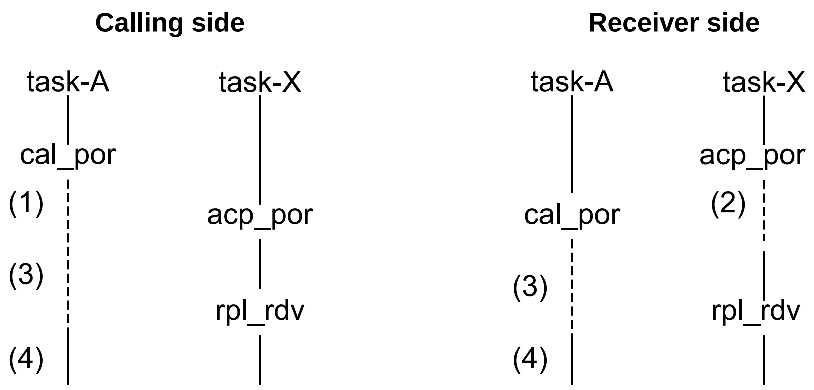
Creation and deletion of rendezvous port can be done by cre_por, acre_por and del_por system calls. There is a management function cal_por for rendezvous call, acp_por for rendezvous reception and rpl_rdv for rendezvous reply. pacp_por is a system call to do polling mode reception. Moreover, there is tcal_por/tacp_por for rendezvous call & reception in timeout mode. In addition, there are fwd_por to forward the received rendezvous to another port, ref_por to get the port state reference and ref_dev to refer to the state of rendezvous.

Differences from NORTi3

- Rendezvous call waiting by the order of the task priority was added.
- Extended information was deleted from the rendezvous generation information.
- The timeout time of the tcal_por is “until rendezvous ends” instead of “until rendezvous is formed”. Accordingly pcal_por is also corrected.
- Calling message size was changed to return parameter type from the acp_por function input argument.
- With ref_rdv, it is possible to find the WAITING state of the rendezvous partner.
- Rendezvous reception condition = 0, is treated as E_PAR error.
- Vcre_por name was changed to acre_por.

Fundamental flow for rendezvous port operation

Following figure shows the example of rendezvous operation using task-A and task-X. The dotted line indicates the WAITING state.



When task-A issues rendezvous call cal_por, and if task-X has not yet executed rendezvous acceptance acp_por, then task-A enters the rendezvous calling wait state (1).

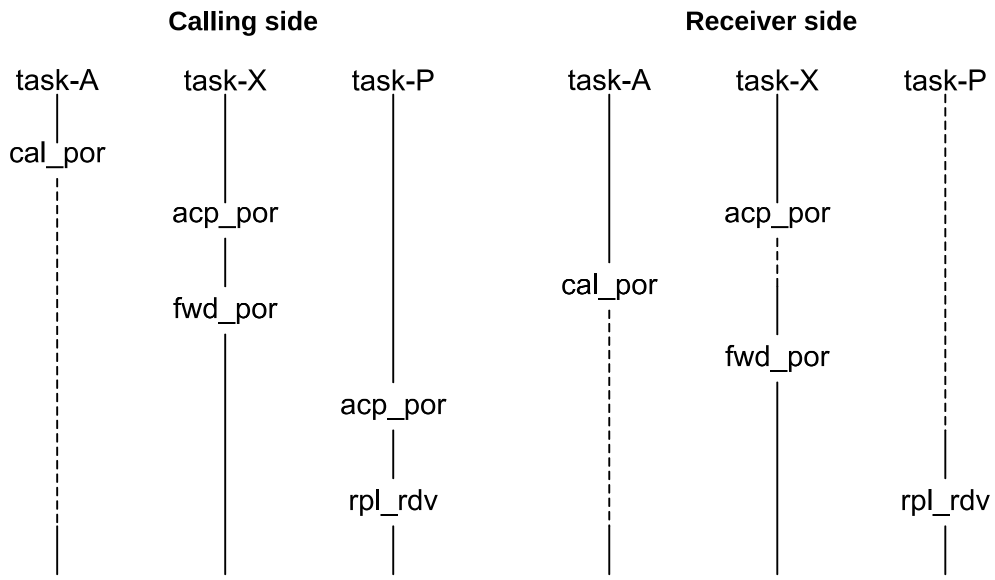
Conversely, when task-X is executing the rendezvous acp_por, and if task-A has not yet issued rendezvous calling cal_por, then task-X enters the wait state (2) to receive rendezvous.

When both calling and accepting are ready, task-A enters the wait state (3) for rendezvous termination. Task-X continues execution and at the point when the rendezvous reply rpl_rdv has been executed, task-A waiting is released (4) and the rendezvous is terminated.

Rendezvous transfer

The received rendezvous can be forwarded to another port by using fwd_por.

The following graph shows an example in which, task-P receives and answers the rendezvous port transfer from task-X.



Conditions for rendezvous operation

Calling side selection condition and receiver side selection conditions can be specified similar to bit pattern of the event flag. A rendezvous is established when a logical AND of the bit pattern of calling side selection condition and the bit pattern of the accepting side selection conditions is non-zero.

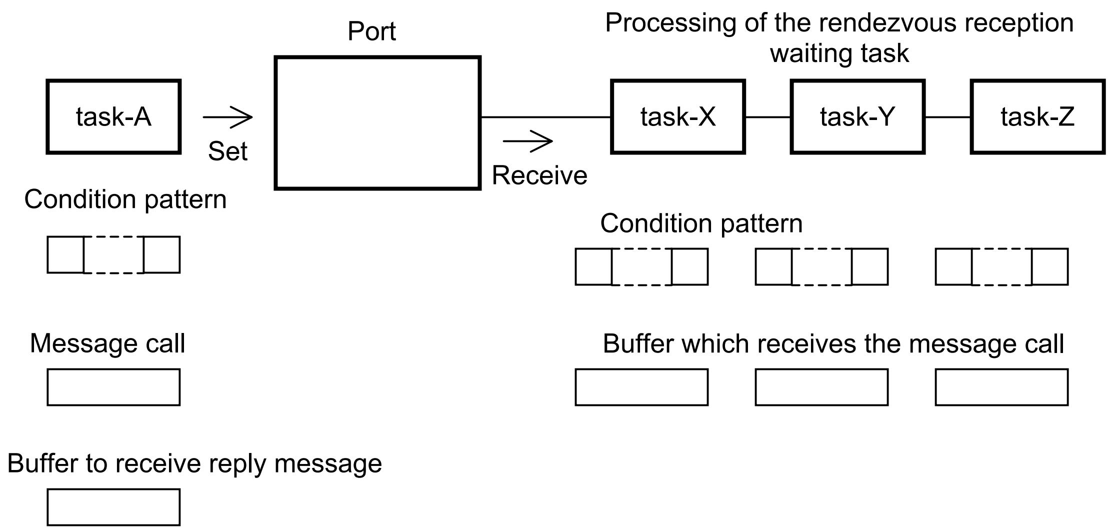
Message

At the time of rendezvous formation, a calling message is passed from the calling task to receiver task. The reply message is passed from the receiver task to the calling task at the time of rendezvous end.

Message is copied between the buffers prepared by respective task. Although the structure resembles to the message buffer function, a message queue does not exist with the type of synchronous method called rendezvous. In addition please note that, since the copy is performed in the state of the interrupt prohibition state, the interrupt prohibition time will get prolonged when a big size data is passed.

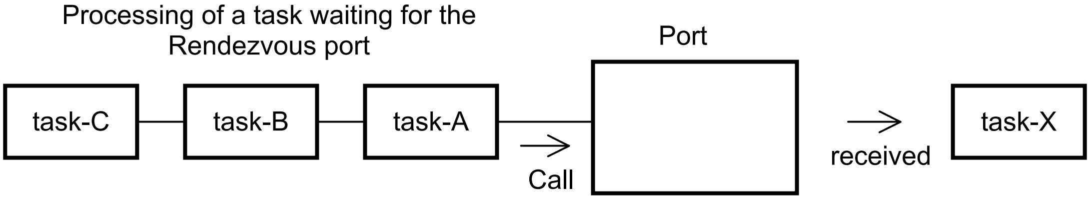
Rendezvous reception waiting queue

Two or more task can wait for rendezvous reception in the same port. In case when there is no calling task or when the redezvous port is not implemented, queuing is built in the order of arrival of reception call. The queuing cannot be made in the order of task priority.



Rendezvous call waiting queue

Two or more tasks can wait for the rendezvous call in the same port. In case when there is no task at receiver side or when rendezvous is not implemented, queuing is built in the order of arrival basis or in the order of task priority.



4.11 Interrupt management function

Overview

The chg_ims, get_ims, ent_int, ret_int, cre_isr, acre_isr, del_isr and dis_int system calls are classified as interrupt management functions. In addition, the def_inh and ena_int system calls are dependent upon implementation (that is, the user can customize it)

Differences from NORTi3

- loc_cpu and unl_cpu were classified into the system state management functions.
- def_int system call name was changed to def_inh.
- ref_ims system call name was changed to get_ims.
- ret_wup system call was removed.
- cre_isr, acre_isr, del_isr and ref_isr are the newly added system calls.

Definition of interrupt handler and interrupt service routine

An interrupt vector is set up using system call def_inh that defines an interrupt handler and system calls cre_tsk, acre_tsk and del_isr. But the method of setting up the interrupt defers according to the system and so such a system call is not included in the kernel. If the system call defined in the attached n4ixxxx.c does not match, the user need to set up an original function.

Prohibiting and permitting individual interrupt

The dis_int and ena_int system calls prohibit or permit particular interrupts in the μITRON 4.0 specification, but depend completely on implementation. In NORTi none of the processes support these system calls (They might be contained in samples for processors that can create general-purpose dis_int and ena_int.).

Start of Interrupt handler

The kernel does not process interrupt before the interrupt handler. It flies directly to the interrupt handler described by the user.

NORTi sets up an ent_int system call, as a unique specification. This is called at the entry of the interrupt handler, so that interrupt handlers are all described in C. The ent_int system call not only saves all registers but also changes stack pointers to stack ranges dedicated for interrupt handlers.

Start of interrupt service routine

When an interrupt, which has registered interrupt service routine, is generated, the interrupt handler is first controlled by the kernel and then the user defined interrupt service routine is executed.

RISC processor interrupt

In RISC processors like ARM, MIPS, PowerPC, SH-3/4, and so on, all the outer interrupts have a common single point entry. In this case, in a def_inh system call, the address of an interrupt handler is to set in the arrangement defined in n4ixxx.c instead of an interrupt vector table. In addition, the program, which distinguishes interrupt factors and jumps referring to this arrangement, is described as a sample in vecxxx.asm. (initarm.xxx in case of ARM processor) Therefore the RISC processor based system can also be programmed as if there is an interrupt vector table. The permission/prohibition properties about system calls, ent_int and ret_int are same as the case about CISC processor.

Interrupt routine of priority higher than kernel

The interruption routine of level higher than the interrupt-inhibit level of a kernel can be used. For this interruption routine, the interrupt-inhibit section inside a kernel becomes the same thing as interruption permission, and NORTi can be applied now also by the system by which a very high-speed interrupt acknowledgement is demanded

However, by the interruption routine of high priority, a system call cannot be published from a kernel. Instead of register bank copy and restoration at the entry and exit of interrupt in ent_int() and ret_int(), please perform the interrupt function offered by compiler or code it using the assembly.

In the interruption routine of high priority, a synchronization or communication with a task cannot be performed from a kernel. When it is necessary to synchronize and communicate with a task by the break of a series of interruption, please start the interrupt handler below the level of a kernel from the interruption routine of high priority, and use the program which uses a system call there.

4.12 Memory pool management function

Overview

Memory pool management functions in the compact NORTi OS offer handling with fixed-length memory block and variable length memory block. Create a program such that when memory is necessary, a memory block is acquired from the memory pool and it is returned to the same memory pool when not needed. The memory area shared among tasks is controlled in units called memory pool. One memory pool consists of more than one memory block.

Memory pool functions are similar to malloc / free functions in standard C libraries. Memory pool functions differ from malloc/free functions, as the former possess functions appropriate to multi-tasking, such as releasing waits for memory acquisition of other tasks when memory is released.

Memory pools with fixed length are created with cre_mpf and acre_mpf. Contrasting with the rel_mpf system call that returns a memory block, the get_mpf system call waits for acquisition of a memory block, the pget_mpf system call polls without waiting, and the tget_mpf system call waits with a time-out. Besides these, the ref_mpf system call references the conditions of fixed-length memory pools.

Differences from NORTi3

The names of rel_blk, get_blk, pget_blk, tget_blk system call were changes to xxx_mpl respectively.

The names of rel_blf, get_blf, pget_blf, tget_blf were changed to xxx_mpf.

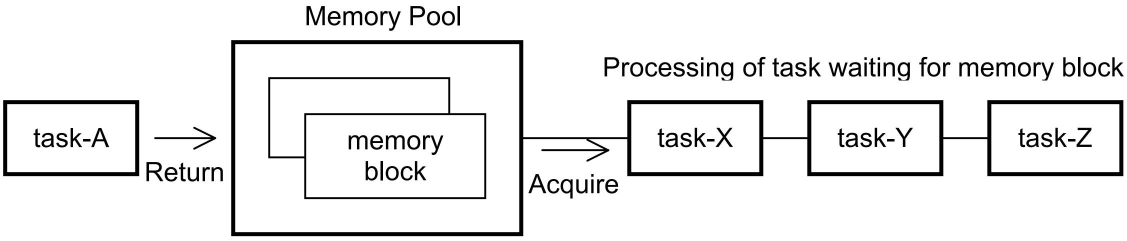
The names of vcre_mpl, vcre_mpf system calls were changed to acre_xxx.

Extended information was deleted from the creation information data structure.

Extended information was removed from the reference information from ref_mpl and ref_mpf.

Memory block waiting queue

Two or more tasks can wait for same memory pool. When FIFO is specified at the time of memory pool generation, queuing is built in othe order of arrival. In case when the task priority order is specified, queuing is built in the order of priority of task (in the order of arrival among task with same priorities).



Though the chart above shows both the tasks waiting for memory block and memory block itself, they do not exist at the same time in the fixed-length memory pool.

Combination with sending and receiving messages

Generally, the memory block in a memory pool is used for the message packet range in mailbox functions. Users must program the memory block to be acquired on the sending message side and to be returned to the receiving message side.

Variable length and fixed length

Since the variable length memory pool is processed using dynamic memory management, it is more convenient than the fixed length memory pool. Variable length memory pool is suitable for high scale system. It is recommended to use the fixed length memory pool when a system can be managed with fixed size memory.

In case of the variable length memory pool, when 1 memory block is acquired, int size memory is used to maintain memory pool size information. With fixed length memory pool there is no useless memory consumption.

Multiple memory pools

It is recommended to provide more than one memory pool for each application. Using only one memory pool for various tasks can cause deadlock when the memory pool becomes empty. In other words, delay at one place may affect the entire system, causing a processing failure.

For example, assume that task A, task B, and task C operate in cooperation with message transmission/reception in which memory pools are combined. As a flow of processing, assume that task A sends a command message to task B, and that the task B that has received it also sends the command message to task C, then the task C that has received it sends back a reply message to task B. If task C is slow in processing, messages from task A to B are consecutively queued, and at last the memory block has all been used up. It causes task C that has terminated processing to be incapable of acquiring a memory block to return a reply message. Further, task B waiting for the reply stops permanently.

On the other hand, dividing the memory pools for each application allows positive use of an empty memory pool. Thus the number of processes being queued can be controlled.

4.13 Time management functions

Overview

The set_tim, get_tim, cre_cyc, acre_cyc, del_cyc, sta_cyc, stp_cyc, ref_cyc, cre_alm, acre_alm, del_alm, sta_alm, stp_alm, ref_alm, def_ovr, sta_ovr, stp_ovr, ref_ovr, isig_tim system calls are classified as a time management functions.

Differences from NORTi3

- Apart from the system-clock used by the system, a system-time was introduced for user.
- Specifications of set_tim and get_tim system calls were changed in order to set up and refer to the system time.
- Overrun handler was introduced to monitor task execution time.
- The function to handle the starting phase is added to the cyclic handler.
- cycact was deleted from the cyclic handler gerenation information. It is in the stop state at the time of generation.
- The function to start alarm handler at absolute time was removed.
- alarm handler release is not performed automatically.
- act_cyc was divided into sta_cyc and stp_cyc.
- def_cyc was divided into cre_cyc and del_cyc.
- acre_cyc was added newly.
- def_alm system call was changed to cre_alm.
- del_alm system call was added newly.
- sta_alm and stp_alm were added newly.
- ret_tmr was removed.

System time and system clock

System clock is reset to 0 at the time of system start after which the clock-count increments for every cyclic interrupt.

System time can be changed using the set_tim system call and after that the count rises with every periodic interrupt. This system time value can be read by get_tim. System time is undefined until it is set by set_tim.

Since a timer event handlers are started with the system clock as the base, even if the system time is changed, it does not affect the previously set up timer event operations.

The timer interrupt cycle is set as the unit of the time so as to avoid unnecessary overhead of multiplication and division inside the system call.

Cyclic handler

A cyclic handler is a time event handler, which is activated periodically at the specified time. Using cyclic handler it is possible to sample data that demands interval time accuracy, or implementation of round-robin type scheduling ring by using rot_rdq etc.

A cyclic handler is created using cre_cyc or acre_cyc and can be deleted using del_cyc system call. In addition, there is ref_cyc system call which referes to the cyclic handler state, the system calls sta_cyc to start and stp_cyc to stop the cyclic handler.

Alarm handler

This time event handler is executed only once after the specified time is expired.

An alarm handler can be registered by cre_alm or acre_alm system call and can be cancelled by del_alm. The activation time of the alarm handler is not set at the time of creation. The alarm handler activation time is set by sta_alm service call and it can be stopped by stp_alm service call. Setup cancellation is not done although it is cancelled automatically when an alarm handler is started. To find out the state of the alarm handler, ref_alm system call is available.

Overrun handler

Overrun handler is a time event handler, which is activated when a task is executed for a time longer than the set time. System clock is used to monitor the task processing time. For this reason, the overrun handler time monitoring is not accurate when a set time is below system clock interval time or when it is not perfect multiple of system clock interval time. Overrun handler is useful to monitor the task that may go into infinite loop depending on the processing conditions.

Only one overrun handler can be defined to whole system using def_ovr system call. When sta_ovr system call is invoked to start the overrun handler, a system time is setup for the specified task. Stp_over system call is used to stop/cancel the overrun handler. It is possible to setup overrun handler for two or more tasks. The operational state of the overrun handler and the remaining processor time can be referred from ref_ovr system call.

4.14 Extended service call management function

Overview

A service call management function does the definition and a call of an extended service call. An extended service call is a function for call a module when a system does not have all modules such as the module loaded dynamically, module put on the firmware etc. are linked together.

Registration / release of the extended service call can be done with def_svc. Extended service call calls the routine registered by cal_svc.

Differences from NORTi3

This function is introduced from μITRON4.0

Extended service call routine description

```
ER_UINT svcrtn(VP_INT par1, VP_INT par2,…, VP_INT par6)
{
        :
        :
}
```

Please describe the service call routine in C language as shown above. 0 ~ 6 parameters can be specified with the service call routine.

4.15 System state management function

Overview

The system state management functions used to refer or change the system management are, rot_rdq (for rotating ready queue), get_tid, vget_tid (to obtain task ID of the self task), loc_cpu, uloc_cpu (to lock / unlock the CPU), dis_dsp, ena_dsp (to disable / enable the dispatch), sns_loc, sns_ctx, sns_dsp, sns_dpn, ref_sys (system call reference functions).

Differences from NORTi3

It is a new functional category to NORTi4.

When a get_tid is called from the non-task context, instead of FALSE, ID of RUNNING task is returned.

CPU lock state and dispatch prohibition state are made independent.

Control of the order of task execution

As per dis_dsp (dispatch disable) and ena_dsp (dispatch enable), when two or more system calls are issued, task switching can be performed collectively. As per rot_rdq (rotate ready queue), it is possible to control the order among task of same priority as round-robin style.
Interrupts are temporarily forbidden when CPU is locked.

4.16 System configuration management functions

The system call ref_ver (OS version reference) and ref_cfg (refers to the configuration information), are classified into a System Management Function.

Differences from NORTi3

get_ver system call name was changed to ref_ver.

Un-supported functions

CPU exception handler definition function (def_exc) is not supported in NORTi.

※ From the next page onwards error types are classified as below.

In the system call description in next chapter, the * and ** mark indicators are defined as below.

* In the library without parameter check, static error is not outputted.
In the standard library, error check is updated in SYSER variable.

** In all libraries, SYSER variable is always updated.

When none of above mark is there, the SYSER error variable is not updated in system library.

5. System Call Description

5.1 Task management functions

cre_tsk

Function Task creation

Declaration ER cre_tsk(ID tskid, const T_CTSK *pk_ctsk);

tskid	Task ID
pk_ctsk	Task creation information packet pointer

Description The cre_tsk system call creates tasks specified by tskid. That is, it dynamically allocates a task management block (TCB) from system memory. In addition, it dynamically allocates the stack area from stack memory when the stack domain start address of the task generation information packet is NULL. As a result of creation, the object task transfers from the NON-EXISTENT state to the DORMANT state.

The structure of the task generation information packet is as follows.

```
Typedef struct t_ctsk
{      ATR tskatr;          Task attribute
       VP_INT exinf;        Extended Information
       FP task;             Function pointer for the task
       PRI itskpri;         Priority at the time of task starting
       SIZE stksz;          Stack size (in bytes)
       VP stk;              Stack domain start address
       B *name;             Task name pointer (optional)
} T_CTSK;
```

The value of exinf is passed to the task as the task parameter when task is started by act_tsk. In addition, exinf value is also passed to an overrun handler. exinf can be referred by ref_tsk system call.

Please specify tskatr as TA_HLNG that shows that task is described in high-level language. Moreover, please specify TA_ACT when a state transition from DORMANT state to READY state is required after task creation.

Please specify name as the task name character string. OS does not use name as an object or for debugger. Please specify “” or NULL as default specification. You may omit name when T_CTSK object structure is defined with an initial value.
When a stack memory domain is reserved in the user program, please set stack head address to stk and set the stack size to stksz parameters.

Return	E_OK	Normal end
	E_PAR	Task priority is outside range*
	E_ID	Task ID is outside range*
	E_OBJ	The task is already generated.
	E_CTX	Issued from an interrupt handler.
	E_SYS	Failed to allocate memory for a management block. *
	E_NOMEM	Failed to allocate stack memory.

Notes

As the task generation information packet is not copied to the task management block, you must keep it even after this system call has been issued. Please define it as a const variable and place it in the ROM domain. If it is placed in domain other than ROM, then a copy of task generation information packet is created in the system memory in order to prevent abnormal operations due to changes or damage during program execution.

Example

```
#define ID_task2    2
const T_CTSK ctsk2 = {TA_HLNG, NULL, task2, 8, 512, NULL};

TASK task1(void)
{
    ER ercd;
        :
    ercd = cre_tsk(ID_task2, &ctsk2);
        :
}
```

acre_tsk

Function Task creation (automatic ID allocation)

Declaration ER_ID acre_tsk(const T_CTSK *pk_ctsk);
pk_ctsk Task creation information packet pointer

Description The acre_tsk system call allocates highest ID from the non-generated task Ids. When no task ID is allocated, the system call returns an E_NOID error. Otherwise, this is the same as cre_tsk.

Return After successful operation, a positive ID value is returned.

E_PAR	A priority is outside valid range*
E_NOID	Insufficient task ID
E_CTX	The command issued from an interrupt handler*
E_SYS	Could not allocate memory for management block**
E_NOMEM	Insufficient stack memory**

Example

```
ID ID_task2;
const T_CTSK ctsk2 = {TA_HLNG, NULL, task2, 8, 512, NULL};

TASK task1(void)
{
    ER_ID ercd;
        :
    ercd = acre_tsk(&ctsk2);
    if(ercd > 0)
        ID_task2 = ercd;
        :
}
```

del_tsk

Function Task deletion

Declaration ER del_tsk(ID tskid);

tskid Task ID

Description The del_tsk system call deletes tasks specified by tskid. It releases the stack range for this task back to stack memory and releases the task control block (TCB) back to system memory. As a result of deletion, the object task transfers from the DORMANT state to the NON-EXISTENT state. Please use exd_tsk to delete self-task, as the task itself cannot specify this system call.

Return

E_OK	Successful termination
E_ID	Task ID is outside valid range*
E_OBJ	Self-Task specification (tskid = TSK_SELF)*
E_CTX	The command issued from an interrupt handler*
E_NOEXS	Task do not exist
E_OBJ	Task is not in DORMANT state

Note Resources other than mutex that an object task acquires (such as memory blocks and semaphores) are not released automatically. Users are responsible for releasing resources before deleting tasks.

Example

```
#define ID_task2    2

TASK task1(void)
{
            :
    del_tsk(ID_task2);
            :
}
```

act_tsk
iact_tsk

Function Task starting

Declaration

ER act_tsk(ID tskid);
ER iact_tsk(ID tskid);

tskid	Task ID

Description

This system call starts tasks specified by tskid. Iact_tsk is the macro re-definition of act_tsk, for compatibility with μITRON specifications. The object task transfers from the DORMANT state to the READY state (When this task has higher priority than the current running task, it transfers to the RUNNING state). When the object task is not in the DORMANT state, this system call queues start requests. The extended information contained in the information for task creation, is passed to task handler at the time the task starts.

If TSK_SELF is specified in tskid, it becomes the start request for the task itself and is queued.

Return

E_OK	Successful termination
E_ID	Task ID is outside valid range*
E_NOEXS	Task do not exist
E_QOVR	Queue overflow

Example

```
#define ID_task2    2
#define ID_task3    3
const T_CTSK ctsk2 = {TA_HLNG, 1, task2, 8, 512, NULL};
const T_CTSK ctsk3 = {TA_HLNG, NULL, task3, 8, 512, NULL};

TASK task2(int exinf)
{
        if(exinf == 1)
                :
}

TASK task3(void)            /* When exinf is not used */
{
                :
}
```

```
TASK task1(void)
{
        :
    cre_tsk(ID_task2, &ctsk2);
    cre_tsk(ID_task3, &ctsk3);
        :
    act_tsk(ID_task2);
    act_tsk(ID_task3);
        :
}
```

can_act

Function Cancellation of task start request

Declaration ER_UINT can_act(ID tskid);

tskid Task ID

Description This system call cancels the request to start a task specified by tskid and makes it 0.

A self-task can be specified with tskid = TSK_SELF.

Return When it is 0 or positive value, it indicates the number of start requests in the queue (actcnt).

E_ID Task ID is outside valid range*

E_NOEXS Task do not exist

Example

```
#define ID_task2    2
const T_CTSK ctsk2 = {TA_HLNG, 1, task2, 8, 512, NULL};

TASK task2(int exinf)
{
            :
}

TASK task1(void)
{
        cre_tsk(ID_task2, &ctsk2);
            :
        act_tsk(ID_task2);
            :
        can_act(ID_task2);
            :
}
```

sta_tsk

Function Tak starting

Declaration ER sta_tsk(ID tskid, VP_INT stacd);

tskid	Task ID
stacd	Task starting code

Description The sta_tsk system call starts tasks specified by tskid and passes stacd (when stacd is not used, 0 is passed). The object task transfers from the DORMANT state to the READY state (when this task has higher priority than the currently running task, it transfers to the RUNING state).

Start demands by this system call are not queued. Accordingly, when the object task is not in the DORMANT state, an error is returned.

Return

E_OK	Successful termination
E_ID	Task ID is outside valid range*
E_OBJ	Self-task specification (tskid = TSK_SELF)*
E_NOEXS	Task do not exist
E_OBJ	The task is readly started

Example

```
#define ID_task2    2
#define ID_task3    3

TASK task2(int stacd)
{
        if (stacd ==1)
                :
}

TASK task3(void)            /* When stacd is not used */
{
                :
}

TASK task1(void)
{
                :
        sta_tsk(ID_task2, 1);
        sta_tsk(ID_task3, 0);
                :
}
```

ext_tsk

Function Terminate self-task

Declaration void ext_tsk(void);

Description By this system call, a task terminates by itself. If there is no start demand in queue, the task transfers from the RUN state to the DORMANT state. When the start requests are in queue, it task is restarted after reducing the queue count by 1. The internal state of the task is initialized during restart. In other words, the task unlocks the mutex, cancels the overrun handler registration, blocks the task-exeption handler and resets the values for priority, wakeup requests, forced wakeup requests, suspend/resume factors and stack.

After restart, the task is connected to the tail of initial priority ready queue.

Return None (it does not return to calling function)

Note Following error is detected internally.

E_CTX Issued from non-task context or in dispatch prohibition state*

Any resources other than mutex that have been acquired by the task (such as memory blocks and semaphores) are not released automatically. Users are responsible for releasing resources before terminating the task.

Example

```
TASK task2(void)
{
            :
    ext_tsk();
}
```

Even if this function is not called clearly as above, it is automatically called by the return from the main routine.

exd_tsk

Function Terminate and delete the self-task.

Declaration void exd_tsk(void);

Description By this system call, a self-task is terminated and then deleted. The call releases the stack domain for this task back to stack memory and releases the task control block (TCB) back to system memory. As a result of deletion, the task changes from the RUNNING state to the NON-EXISTENT state. Any start request in the queue will be cancelled.

Return None (it does not return to calling function)

Note Following error is detected internally.

E_CTX Issued from non-task context or in dispatch prohibition state*

Any resources other than mutex that have been acquired by the task (such as memory blocks and semaphores) are not released automatically. Users are responsible for releasing resources before terminating the task.

Example

```
TASK task2(void)
{
            :
    exd_tsk();
}
```

ter_tsk

Function Remote task forced termination.

Declaration ER ter_tsk(ID tskid);

tskid Task ID

Description The ter_tsk system call terminates the task specified by tskid. As a result of termination, the object task transfers from the READY, WAITING or WAITING-SUSPEND state to the DORMANT state. When the start requests are queued, it restarts. When the object task is connected to a waiting queue, executing ter_tsk removes the object task from the queue. Self-task ID cannot be specified to this system call.

Return

E_OK	Successful termination
E_ID	Task ID is outside valid range*
E_ILUSE	Self-task specification (tskid = TSK_SELF)*
E_NOEXS	Task do not exist
E_OBJ	Task is not yet started

Note Any resources other than mutex that have been acquired by the task (such as memory blocks and semaphores) are not released automatically. Users are responsible for releasing resources before terminating the task.

Example

```
#define ID_task2    2

TASK task1(void)
{
            :
    ter_tsk(ID_task2);
            :
}
```

chg_pri

Function Change the task base priority

Declaration ER chg_pri(ID tskid, PRI tskpri);

tskid	Task ID
tskpri	Task priority to set

Description The chg_pri system call uses tskpri values for the priority of the task specified by tskid. The smaller the number, the higher the priority. There are three priorities i.e. initial priority, base priority and current priority. Initial priority is the priority specified at the time of task creation (itskpri) and is set as base priority value when task starts. And this is copied to base priority when the task starts. Tasks are normally run by base priority but when mutex is locked, the priorities change temporarily. This changed priority is the current priority. When mutex is unlocked, the task priority goes back to base priority. Chg_pri changes that base priority. Usually a task runs with a base priority, but priority may change temporarily when a mutex is locked. The priority changed temporarily is the present priority. After mutex is unlocked, the task priority changes back to the base priority.

A self-task can be specified with tskid=TSK_SELF. Tskpri=TPRI_INI specifies the initial priority, tskpri=TMIN_PRI indicates the maximum priority nad tskpri=TMAX_PRI specifies the minimum priority

When the object tasks are queued (ready queue, semaphore or memory pool waiting queue etc.) in the order of priority, the queuing of waiting connections is rearranged by change in the priority. The waiting connections in the queue are rearranged even when the current priority is changed. Please note than when mutex is used, waiting connections in the queue are exchanged dynamically.

When the priority of an object task in the READY state is made higher than the priority of a host task, which issued this system call, then the task issuing this system call transfers from the RUNNING state to the READY state and the object task transfers to the RUNNING state.

When the priority of the self-task is made lower than other READY tasks, then the self-task changes from the RUNNING state to the READY state and the task with the highest priority among the other READY tasks will move to the RUNNING state.

When the same priority as that of the present task is specified, and if there exists other tasks with same priority, the object task goes to the tail of the priority queue.

The priority changed by this system call is effective until tasks are terminated. When the task restarts, the task priority returns to initial priority.

Return	E_OK	Successful termination
	E_PAR	Priority is outside valid range*
	E_ID	Task ID is outside valid range*
		TSK_SELF is specified in the non-task context *
	E_NOEXS	Task do not exist
	E_OBJ	Task is not yet started

Example

```
TASK task1(void)
{
        :
    chg_pri(TSK_SELF, TMIN_TPRI);   /* temporarily set to the highest priority */
        :
    chg_pri(TSK_SELF, TPRI_INI);    /* return back to the base priority */
        :
}
```

get_pri

Function Refer to the current task priority

Declaration ER get_pri(ID tskid, PRI *tskpri);

tskid	Task ID
tskpri	memory pointer to store the current task priority

Description This system call returns the current priority of the task specified by tskid.
A self-task can be specified with tskid = TSK_SEL.F.

Return

E_OK	Successful termination
E_ID	Task ID is outside valid range*
E_NOEXS	Task do not exist
E_OBJ	Task is not yet started

Example

```
TASK task1(void)
{
        PRI tskpri;
                :
        get_pri(TSK_SELF, &tskpri);
                :
}
```

ref_tsk

Function Refer to the task state

Declaration ER ref_tsk(ID tskid, T_RTSK *pk_rtsk);

tskid	Task ID
pk_rtsk	memory pointer to the task state packet

Description The state of the task specified by tskid, is returned to *pk_rtsk.

A self-task can be specified by tskid=TSK_SELF.

Following is the task state packet structure.

```
Typedef struct t_rtsk
{   STAT tskstat;     Task state
    PRI tskpri;       Current priority
    PRI tskbpri;      Base priority
    STAT tskwait;     Waiting factor
    ID wid;           ID of waiting object
    TMO lefttmo;      Left time until timeout
    UINT actcnt;      Start request count
    UINT wupcnt;      Wakeup request count
    UINT suscnt;      Suspend request count
    VP exinf;         Extended information
    ATR tskatr;       Task attribute
    FP task;          Task handler start address
    PRI itskpri;      Initial priority at the time of task starting
    int stksz;        Stack size (byte count)
}T_RTSK;
```

The values specified by task generation returns to exinf, tskatr, task, itskpri, & stksz parameters as it is.

Following values are returned to the task state parameter, tskstat.

TTS_RUN	0x0001	RUNNING State
TTS_RDY	0x0002	READY State
TTS_WAI	0x0004	WAITING State
TTS_SUS	0x0008	SUSPENDED State
TTS_WAS	0x000c	WAITING-SUSPENDED State
TTS_DMT	0x0010	DORMANT State

When the task is in WAITING state, the following values are returned to the task state parameter, tskwait.

TTW_SLP	0x0001	Waiting by slp_tsk or tslp_tsk
TTW_DLY	0x0002	Waiting by dly_tsk
TTW_SEM	0x0004	Waiting by wai_sem or twai_sem
TTW_FLG	0x0008	Waiting by wai_flg or twai_flg
TTW_SDTQ	0x0010	Waiting by snd_dtq
TTW_RDTQ	0x0020	Waiting by rcv_dtq
TTW_MBX	0x0040	Waiting by rcv_msg or trcv_msg
TTW_MTX	0x0080	Waiting by loc_mtx
TTW_SMBF	0x0100	Waiting by snd_mbf or tsnd_mbf
TTW_RMBF	0x0200	Waiting by rcv_mbf or trcv_mbf
TTW_CAL	0x0400	Waiting for a rendezvous call
TTW_ACP	0x0800	Waiting for a rendezvous reception
TTW_RDV	0x1000	Waiting for a rendezvous end
TTW_MPF	0x2000	Waiting for acquisition of fixed length memory block
TTW_MPL	0x4000	Waiting for acquisition of variable length memory block

Return

E_OK	Successful termination
E_ID	Task ID is outside valid range
E_NOEXS	Task do not exist

Example

```
#define ID_task2   2

TASK task1(void)
{
    T_RTSK rtsk;
        :
    ref_tsk(ID_task2, &rtsk);
    if(rtsk.tskstat == TTS_WAI)
        :
}
```

ref_tst

Function Refers to the task state

Declaration ER ref_tst(ID tskid, T_RTST *pk_rtst);

tskid	Task ID
pk_rtst	memory pointer to the task state packet

Description The state of the task specified by tskid, is returned to *pk_rtst.

A self-task can be specified by tskid=TSK_SELF.

Following is the task state packet structure.

```
Typedef struct t_rtst
{       STAT tskstat;           Task state
        STAT tskwait;           Wait factor
}T_RTST;
```

The parameters tskstat, tskwait returns the same contents as described in ref_tsk.

Return

E_OK	Successful termination
E_ID	Task ID is outside valid range
E_NOEXS	Task do not exist

Example

```
#define ID_task2    2

TASK task1(void)
{
        T_RTSK rtst;
                :
        ref_tst(ID_task2, &rtst);
        if (rtst.tskstat == TTS_WAI)
                :
}
```

5.2 Task associated synchronization functions

sus_tsk

Function Task suspend (compulsory waiting state)

Declaration ER sus_tsk(ID tskid);

tskid Task ID

Description The sus_tsk system call suspends the execution of tasks specified by tskid. When the object task is in the READY state, the system call transfers it to the SUSPENDED state. When the object task is in the WAITING state, the system call transfers it to the WAITING-SUSPEND state. A self-task can be specified by tskid=TSK_SELF.

This suspended task can be released by the rsm_tsk or frsm_tsk system call. Task suspend commands can be nested, i.e. when rsm_tsk is issued for the same number of times as sus_tsk is issued, then a SUSPENDED state is released for the first time.

Return

E_OK	Successful termination
E_ID	Task ID is outside valid range*
E_CTX	Self-task is specified in dispatch prohibition state (tskid=TSK_SELF)*
E_NOEXS	Task do not exist
E_OBJ	Task is not yet started
E_QOVR	Suspend request wait queue overflow (TMAX_SUSCNT exceeded 255)

Example

```
#define ID_task2    2

TASK task1(void)
{
            :
    sus_tsk(ID_task2);
            :
}
```

rsm_tsk

Function	Resume the task from the suspended state

Declaration ER rsm_tsk(ID tskid);

tskid	Task ID

Description The rsm_tsk system call releases the suspended execution of the task specified by tskid. When the object task is in the SUSPENDED state, it transfers to the READY state (When the object task has priority higher than the present running task, it transfers to the RUNNING state). When the object task is in the WAIT-SUSPENDED state, it transfers to the WAITING state.

Rsm_tsk system call releases single sus_tsk request. In other words, when sus_tsk is issued more than once, the object task remains in the SUSPENDED state after rsm_tsk is executed.

A self-task cannot be specified in this system call.

Return

E_OK	Successful termination
E_ID	Task ID is outside valid range*
E_OBJ	Self-task specification (tskid = TSK_SELF)*
E_NOEXS	Task do not exist
E_OBJ	Task is not in SUSPENDED state

Example

```
#define ID_task2    2

TASK task1(void)
{
            :
    sus_tsk(ID_task2);
            :
    rsm_tsk(ID_task2);
            :
}
```

frsm_tsk

Function	Resume the task forcibly from the suspended state

Declaration ER frsm_tsk(ID tskid);

tskid	Task ID

Description The frsm_tsk system call forcibly releases the suspended execution of the task specified by tskid. When the object task is in the SUSPENDED state, it transfers to the READY state (When the object task has priority higher than the present running task, it transfers to the RUNNING state). When the object task is in the WAIT-SUSPENDED state, it transfers to the WAITING state.

Frsm_tsk system call releases all suspend command from the queue. In other words, when sus_tsk is issued more than once, the object task is released from SUSPENDED state after frsm_tsk is executed once.

Return

E_OK	Successful termination
E_ID	Task ID is outside valid range*
E_OBJ	Self-task specification (tskid = TSK_SELF)*
E_NOEXS	Task do not exist
E_OBJ	Task is not in SUSPENDED state

Example

```
#define ID_task2    2

TASK task1(void)
{
            :
        sus_tsk(ID_task2);
        sus_tsk(ID_task2);
            :
        frsm_tsk(ID_task2);
            :
}
```

slp_tsk

Function Sleep the local task

Declaration ER slp_tsk(void);

Description A task transfers itself to the WAITING state. This WAITING state is released by issuing the wup_tsk or rel_wai system call.

When wup_tsk is issued first i.e.when wake up request is queued, slp_tsk does not put the task in wait state. In this case, system call decrements the wake up request count by 1 and then the call returns with E_OK as normal termination return value. The ready queue for the task does not change at this time.

When a task is released by rel_wai, the call returns an E_RLWAI error.

Return

E_OK	Normal End.
E_CTX	Wait at the task independent section or dispatch prohibited state*
E_RLWAI	The wait state has forcibly released (rel_wai was accepted during the wait.)

Note It is same as tslp_tsk(TMO_FEVR)

Example

```
#define ID_task1    1

TASK task1(void)
{
            :
      slp_tsk();
            :
}

TASK task2(void)
{
            :
     wup_tsk(ID_task1);
            :
}
```

tslp_tsk

Function	Sleep the local task (Timeout available)

Declaration ER tslp_tsk(TMO tmout);

tmout	Timeout value

Description A task transfers itself to the WAITING state. This WAITING state is released by issuing the wup_tsk or rel_wai system call for this task, or after termination of time specified by tmout.

When a wait is released by wup_tsk, the tslp_tsk system call returns E_OK as normal termination. When wup_tsk is issued first and the wake up request is queued, slp_tsk does not put the task in wait state. The wake up request count is reduced by 1 and the task returns E_OK as normal termination. Ready queue of the task does not change at this time.

When a task is released by rel_wai, the tslp_tsk system call returns an E_RLWAI error. When a task is released by timeout of a specified time, this system call returns an E_TMOUT error. Tmout is measured in units of the system clock interrupt cycle time.

When tmout is set to TMO_POL (=0) and when wakeup requests are queued, then this system call returns immediately with an E_OK return value for normal. It returns an E_TMOUT time-out error code if there is no wakeup request in queue. This system call does not execute timeout by tmout=TMO_FEVR (= -1), i.e. in such case it operates in the same way as slp_tsk.

Return

E_OK	Normal End.
E_CTX	Wait at the task independent section or dispatch prohibited state*
E_RLWAI	Waiting state was released forcibly (rel_wai was issued while waiting)
E_TMOUT	Timeout

Note1 By using NORTi's unique MSEC macro, this system call can be described with waiting time specified in milli second units i.e. tslp_tsk(100/MSEC);
The MSEC macro is defined in kernel.h as "#define10". But when separate value need to be applied as system clock, please define the value to all places before kernel.h is included.

Note2

After issuing the system call with timeout, since the timing until the first cycle of interrupt timer is attached, there is an error of –MSEC ~ 0 in timeout. For example, for MSEC=10, when a timeout of 100msec is set, a timeout in real time will be in the range of 90msec ~ 100msec. The timeout in μITRON4.0 is specified as the event when time more than the specified timeout time is exceeded. In other words, as in above example a valid timeout is in the range of 100~110msec. In case of NORTi, a valid timeout is in the range of 90~100msec.

Since the task, which performs time waiting, operates in synchronization with periodic timer interrupt, the difference as shown below will occur.

```
For(;;){
    led_on();                   /* LED light ON */
    tslp_tsk(100/MSEC)          /* Wait for 100msec */
    led_off();                  /* LEDlight OFF */
    tslp_tsk(100/MSEC);         /* Wait for 100msec */
}
```

As per NORTi specification, LED blinks with 200msec interval time.

As per μITRON4.0 specification, LED blinks with 220msec interval time.

Example

```
#define MSEC    2
#include "kernel.h"

TASK task1(void)
{
    ER ercd;
        :
    ercd = tslp_tsk(100/MSEC);
    if(ercd == E_TMOUT)
        :
}
```

wup_tsk
iwup_tsk

Function	Wakeup the remote task

Declaration

ER wup_tsk(ID tskid);
ER iwup_tsk(ID tskid);

tskid	Task ID

Description

The wup tsk system call releases a task placed in the WAITING state due slp_tsk or tslp_tsk system call and change the state to READY state (When the task has priority higher than the current running task, it goes to the RUNNING state, and when it is in the WAITING-SUSPENDED state, it transfers to the SUSPENDED state). The object task is specified by tskid. A self-task can be specified from the task-context.

This request for wakeup is queued, when the object task did not performed slp_tsk or tslp_tsk and is not in the WAITING state. The queued request for wakeup becomes effective later when the object task executes either the slp_tsk or tslp_tsk system call. Thus when the wakeup requests are in queue, slp_tsk and tslp_tsk system call decrements wakeup queue count by 1 and then return immediately to the calling function.

Return

E_OK	Normal End.
E_ID	Task ID is outside valid range*
E_ID	Self-task (tskid = TSK_SELF) is specified in non-task context*
E_NOEXS	Task do not exist
E_OBJ	Task is not yet started
E_QOVR	Wakeup request count overflow (TMAX_WUPCNT exceeded 255)

Example

```
#define ID_task1    1

TASK task1(void)
{
            :
    slp_tsk();
            :
}

TASK task2(void)
{
            :
    wup_tsk(ID_task1);
            :
}
```

can_wup

Function Cancel task wakeup request

Declaration ER_UINT wupcnt = can_wup(ID tskid);

wupcnt	Wakeup request count in queue (when positive value)
tskid	Task ID

Description This system call returns the number of wakeup request, which have been queued in a task specified by tskid. At the same time it releases all wakeup requests from queue. A task itself is specified by tskid=TSK_SELF.

When task wake up is carried out periodically, this system call can be used to judge whether a process is completed within the interval time. When wupcnt is non-zero positive value, then it indicates that the previous operation of wake up request has not been completed within the specified time.

Return 0 or positive value indicates the wakeup request count in queue.

E_ID	Task ID is outside valid range*
E_NOEXS	Task do not exist

Example

```
TASK task1(void)
{
    ER_UINT wupcnt;
        :
    slp_tsk();
    wupcnt = can_wup(TSK_SELF);
        :
}
```

vcan_wup

Function Disable all wakeup requests for local task.

Declaration void vcan_wup(void);

Description vcan_wup system call clears the wakeup requests from the queue. This system call is only for the self-task. This system call is unique to NORTi. If it is only about clearing the wakeup request then this system call is faster than can_wup.

Return None

Example

```
TASK task1(void)
{
        :
    vcan_wup();
    tslp_tsk(100/MSEC);
        :
}
```

rel_wai
irel_wai

Function	Remote task release from waiting

Declaration

ER rel_wai(ID tskid);
ER irel_wai(ID tskid);

tskid	Task ID

Description

When a task specified by tskid is in the WAIT state, the rel_wai system call releases it forcibly. An E_RLWAI error returns to the waiting task that was released. When the object task is in the WAITING state, it transfers to the READY state (If the task has priority higher than the present running task, it transfers to the RUNNING state). When the object task is in the WAITING-SUSPENDED state, it transfers to the SUSPENDED state.

When the object task is in the other state i.e. when object task is not in wait state, the object task generates an E_OBJ error. In this case, the state of the object task does not change. In other words, this system call does not queue requests for releasing the wait state.

Return

E_OK	Normal End.
E_ID	Task ID is outside valid range*
E_OBJ	Self-task specification (tskid = TSK_SELF)*
E_NOEXS	Task do not exist
E_OBJ	Task is not in the waiting state

Example

```
#define ID_task2    2

TASK task1(void)
{
            :
    rel_wai(ID_task2);
            :
}
```

dly_tsk

Function	Delay the local task

Declaration ER dly_tsk(RELTIM dlytim);

dlytim	Delay time

Description This call performs the simple time waiting for the task. Although this function is almost same as tslp_tsk(TMO tmout) system call, the time waiting is not released by wup_tsk system call. It is recommended to use dly_tsk instead of tslp_tsk, when task is performing waiting only for time.

The data type of dlytim (delay time), i.e. RELTIM, is a long type similar to TMO of timeout. Unit of delay time is the interval cycle of the system clock.

Return

E_OK	Normal End.
E_CTX	Wait at the task independent section or dispatch prohibited state*
E_RLWAI	Waiting state was released forcibly (rel_wai was issued while waiting)

5.3 Task exception handling functions

def_tex

Function Define a task exception handling routine

Declaration ER def_tex(ID tskid, const T_DTEX *pk_dtex);

tskid	Task ID
pk_dtex	Pointer to the task exception handling routine definition information packet

Description This system call defines the task exception handling routine for the task specified by tskid. When *pk_dtex is specified as NULL pointer, this system call will undefined the task exception handler for the task specified by tskid. Moreover, re-definition is possible if another definition information packet is specified. In re-definition, the exception handling request and exception handling permission / prohibition state is inherited. A self-task is specified when tskid=TSK_SELF.

When a task is restarted, an exception-handling request is cleared and is set to an exception handling prohibition state. Task exception handling routine is undefined when a task is deleted.

Following is the structure of the task exception handler definition information packet.

```
Typedef struct t_dtex
{       ATR texatr;         Task exception handler attribute
        FP texrtn;          Task exception handler starting address
}T_DTEX;
```

Although OS do not notice contents of texatr, in order to maintain compatibility with other OS conforming to μITRON specification, please set TA_HLNG to texatr. When a definition information packet is placed in memory domain other than ROM, a definition information packet is copied to a system memory.

Return

E_OK	Normal End.
E_ID	Task ID is outside valid range*
E_NOEXS	Task do not exist
E_PAR	Parameter error (texrtn == NULL)*

Example

```
#define ID_task1    1

void texrtn(TEXPTN texptn, VP_INT exinf)
{
            :
}

const T_DTEX dtex ={TA_HLNG, (FP)texrtn };

TASK task1(void)
{
            :
      def_tex(ID_task1, &dtex);
            :
}
```

ras_tex
iras_tex

Function Task exception handling demand

Declaration ER ras_tex(ID tskid, TEXPTN rasptn);
ER iras_tex(ID tslid, TEXPTN rasptn);

tskid	Task ID
rasptn	Task exception factor

Description Exception handling factor specified by rasptn is demanded for the task specified by tskid. When an object task is in the waiting state, the exception factor is suspended and the Exception Handling is not permitted until the object task is in the RUNNING state. A self-task can be specified as an object task when tskid = TSK_SELF.

Return

E_OK	Normal End.
E_ID	A task ID is outside valid range.
E_ID	local task specified from non-task context (tskid = TSK_SELF)*
E_NOEXS	The task is not generated
E_OBJ	Task exception handling routine is not defined
E_PAR	rasptn = 0

Example

```
#define ID_task1    1

TASK task1(void)
{
        :
      ras_tex(ID_task1, 1);
        :
      ras_tex(ID_task1, 2);
          :
}
```

dis_tex

Function Disable Task exception handling

Declaration ER dis_tex(void);

Description In a task context, this system call moves the invoking task to the task exception disabled state. When issued from the non-task context such as timer handler, the call returns with E_CTX error code.

Return

E_OK	Successful termination
E_CTX	Context error
E_OBJ	Task exception handling routine is not defined.

Example

```
TASK task1(void)
{
        :
    dis_tex();
        :
}
```

ena_tex

Function Enable task exception handling

Declaration ER ena_tex(void);

Description This system call enables task exception handling when invoked from self-task in task context or from the task in the interrupt handler that is in execution state. This system call returns E_CTX error when called from the timer handler.

If there is a pending exception code then the exception handling routine will be performed when the corresponding task changes into RUNNING state.

Return

E_OK	Successful termination
E_CTX	Context error
E_OBJ	Task exception handling routine is not defined.

Example

```
TASK task1(void)
{
            :
     ena_tex();
            :
}
```

sns_tex

Function Refer to the state of task exception handling state of self-task

Declaration BOOL sns_tex(void);

Description This system call returns TRUE if the self-task or the invoked task is in the task exception handling disabled state, and returns FALSE if the task exception handling is enabled. TRUE is returned if there is no task in the RUNNING state.

Return

TRUE	Disabled
FALSE	Enabled

Example

```
TASK task1(void)
{
            :
    if(sns_tex())
    {
            :
    }
            :
}
```

ref_tex

Function Refer to the state of Task exception handling

Declaration ER ref_tex(ID tskid, T_RTEX *pk_rtex);

tskid	Task ID
pk_rtex	A pointer to a location which stores a task exception handling state reference packet

Description The task exception handling state of the task specified by tskid is returned to *pk_rtex.

A self task can be specified with tskid=TSK_SELF.

Following is the structure of the task exception handling state packet.

```
Typedef struct t_rtex
{     STAT texstat;         The state of the exception handling
      TEXPTN pndptn;        Pending exception code
}T_RTEX;
```

Texstat parameter returns following values.

TTEX_ENA	0x00	Task exception enabled state
TTEX_DIS	0x01	Task exception disabled state

If there is no pending exception request, pndptn=0.

Return

E_OK	Successful termination.
E_ID	Task ID is outside valid range*
E_NOEXS	A specified task does not exist
E_OBJ	Specified task is in DORMANT state, or the task exception handling routine is not defined.

Example

```
#define ID_task2    2

TASK task1(void)
{
      T_RTEX rtex;
            :
      ref_tex(ID_task2, &rtex);
      if (rtex.pndptn != 0)
            :
}
```

5.4 Synchronization / communication functions (Semaphore)

cre_sem

Function Creation of Semaphore

Declaration ER cre_sem(ID semid, const T_CSEM *pk_csem);

semid Semaphore ID

pk_csem semaphore generation information packet pointer.

Description The semaphore object is created with an object ID semid. The semaphore management block memory is dynamically assigned from the system memory. In addition, the semaphore count is set to the initial value specified by isemcnt of semaphore generation information data.

When a semaphore generation information packet is placed in memory domain other than ROM (i.e. when a const data type is not attached), the generation information packet data is copied to the system memory.

Following is the structure of the semaphore generation information packet.

```
Typedef struct t_csem
{     ATR sematr;          Semaphore attribute
      UINT isemcnt;        Semaphore initial count
      UINT maxsem;         Semaphore maximum value
      B *name;             Pointer to a Semaphore name (optional)
}T_CSEM;
```

Please select either of following as the semaphore attribute sematr.

TA_TFIFO Processing of the waiting task is in the order of arriaval (FIFO).

TA_TPRI Processing of the waiting task is in the order of Priority.

Please set maxsem to the number of enabled semaphore resources. The upper limit value that can be set is defined in TMAX_MAXSEM

Since name is an object for debugger correspondence, please specify “” or NULL when not used. When this structure object is defined with initial value, you may omit name.

Return	E_OK	Sucessful termination
	E_PAR	Semaphore maximum is either negative or exceeds 255*, or the initial value of a semaphore is either negative or exceeds maximum value*
	E_ID	Semaphore ID is outside valid range *
	E_OBJ	The semaphore already exists.
	E_CTX	The command issued from an interrupt handler*
	E_SYS	Insufficient system memory for management block**

Example

```
#define ID_sem1    1

const T_CSEM csem1 = {TA_TFIFO, 1, 1};

TASK task1(void)
{
        ER ercd;
                :
        ercd = cre_sem(ID_sem1, &csem1);
                :
}
```

acre_sem

Function Semaphore creation (automatic ID allocation)

Declaration ER_ID acre_sem(const T_CSEM *pk_csem);

pk_csem Semaphore creation information packet pointer.

Description This function searches the highest ID from the unassigned semaphore Ids. When no semaphore ID is allocated, the system call returns an E_NOID error. Otherwise, this is the same as cre_sem.

Return Semaphore ID is returned after successful completion.

E_PAR	Semaphore maximum is either negative or exceeds 255*, or the initial value of a semaphore is either negative or exceeds maximum value*
E_NOID	Semaphore ID is Insufficient
E_CTX	The command issued from an interrupt handler*
E_SYS	Memory insufficient for management block**

Example

```
ID ID_sem1;
const T_CSEM csem1 = {TA_TFIFO, 0, 1};

TASK task1(void)
{
        ER_ID ercd;
                :
        ercd = acre_sem(&csem1);
        if(ercd > 0)
        ID_sem1 = ercd;
                :
}
```

del_sem

Function Semaphore deletion

Declaration ER del_sem(ID semid);

semid Semaphore ID

Description This function delets the Semaphore specified by *semid*, and the semaphore management block memory is released to system memory.

When there is a task waiting for this semaphore, the waiting of the task is cancelled. E_DLT error (since the semaphore is deleted) will be returned by the task which was waiting for this semaphore.

Return

E_OK	Successful termination.
E_ID	Semaphore ID is outside valid range *
E_NOEXS	The semaphore of the specified ID does not exist.
E_CTX	The command issued from an interrupt handler*

Example

```
#define ID_sem1    1

TASK task1(void)
{
            :
      del_sem(ID_sem1);
            :
}
```

sig_sem
isig_sem

Function	Return the semaphore resources
Declaration	ER sig_sem(ID semid); ER isig_sem(ID semid); semid Semaphore ID
Description	This system call increases the semaphore count by one (returning resources), when there are no tasks waiting for semaphores specified by semid. Error E_QOVR is returned when the semaphore count exceeds the maximum value specified at the time of semaphore creation. When tasks are waiting for this semaphore, the sig_sem system call releases the heading task in the queue from waiting, i.e. this system call transfers the task from the WAITING state to the READY state. (When this task has higher priority than the current RUNNING task, this system call transfers it to the RUNNING state, and when it is in the WAITING-SUSPENDED state, the system call transfers it to the SUSPENDED state).

Return	E_OK	Successful termination.
	E_ID	Semaphore ID is outside valid range*
	E_NOEXS	The semaphore of the specified ID does not exist.
	E_QOVR	Semaphore count overflow

wai_sem

Function Semaphore resource acquisition

Declaration ER wai_sem(ID semid);

semid Semaphore ID

Description When the semaphore count specified by semid is more than 1, this system call decreases the semaphore count by 1 (acquiring resources) and returns immediately.

When the semaphore count is 0, the task which issed this system call is queued for waiting this semaphore. In this case, the semaphore count remains 0.

Return

E_OK	Successful termination.
E_ID	Semaphore ID is outside valid range*
E_NOEXS	The semaphore of the specified ID does not exist.
E_CTX	waiting is performed aither from non-task context or it is in the state of dispatch prohibition*
E_RLWAI	Forced release from waiting state (rel_wai was received in between waiting)
E_DLT	The semaphore was deleted while waiting.

Note It is similar to twai_sem(semid, TMO_FEVR)

Example

```
#define ID_sem1    1

TASK task1(void)
{
        :
    wai_sem(ID_sem1);
        :
    sig_sem(ID_sem1);
        :
}
```

pol_sem

Function Semaphore resource acquisition (polling mode)

Declaration ER pol_sem(ID semid);

semid Semaphore ID

Description When the semaphore count specified by semid is more than 1, this system call decreases the semaphore count by 1 (acquiring resources) and returns immediately.

When the semaphore count is 0, the system call does not enter the WAIT state and returns at once with an E_TMOUT error.

Return

E_OK	Successful termination.
E_ID	Semaphore ID is outside valid range*
E_NOEXS	The semaphore of the specified ID does not exist.
E_TMOUT	Polling failure

Note It is same as twai_sem(semid, TMO_POL) system call.

Example

```
if(pol_sem(ID_sem1) == E_OK)
{
            :
        if (pol_sem(ID_sem1) != E_TMOUT)
            :
}
```

twai_sem

Function Semaphore resource acquisition (Timeout available)

Declaration ER twai_sem(ID semid, TMO tmout);

semid	Semaphore ID
tmout	Timeout value

Description When the semaphore count specified by semid is more than 1, this system call decreases the semaphore count by 1 (acquiring resources) and returns immediately. When the semaphore count is 0, the task which issed this system call is queued for waiting this semaphore. In this case, the semaphore count remains 0.

After the time specified by tmout passes, an E_TMOUT time-out error returns. The twai_sem system call does not execute waits by tmout=TMO_POL (=0). It runs in the same way as pol_sem. It does not execute time-outs by tmout=TMO_FEVR (=-1) It runs in the same way as wai_sem.

Return

E_OK	Successful termination.
E_ID	Semaphore ID is outside valid range*
E_NOEXS	The semaphore of the specified ID does not exist.
E_CTX	waiting is performed aither from non-task context or it is in the state of dispatch prohibition*
E_RLWAI	Forced release from waiting state (rel_wai was received in between waiting)
E_DLT	The semaphore was deleted while waiting.
E_TMOUT	Timeout

Example

```
#define ID_sem1    1

TASK task1(void)
{
        ER ercd;
                :
        ercd = twai_sem(ID_sem1, 100/MSEC);
        if (ercd == E_OK)
                :
}
```

ref_sem

Function Refers the state of the Semaphore.

Declaration ER ref_sem(ID semid, T_RSEM *pk_rsem);

semid	Semaphore ID
pk_rsem	Pointer to the semaphore state reference packet

Description This system call returns the state of the semaphore specified by semid to *pk_rsem data pointer.

The semaphore state packet structure is as shown below.

```
Typedef struct t_rsem
{       ID wtskid;          ID of the waiting task. (TSK_NONE if no waiting task)
        UINT semcnt;        current value of the semaphore count
}T_RSEM;
```

When there is a waiting task, wtskid returns the ID of the heading task in the waiting queue.

Wtskid = TSK_NONE, when there is no waiting task.

Return

E_OK	Successful termination.
E_ID	Semaphore ID is outside valid range.
E_NOEXS	The semaphore of the specified ID does not exist.

Example

```
#define ID_sem1    1

TASK task1(void)
{
        T_RSEM rsem;
                :
        ref_sem(ID_sem1, &rsem);
        if (rsem.wtsk != FALSE)
                :
}
```

5.5 Synchronization / communication functions (Event flag)

cre_flg

Function Event flag creation

Declaration ER cre_flg(ID flgid, const T_CFLG *pk_cflg);

flgid Event flag ID

pk_cflg Pointer to event flag generation information packet

Description The cre_flg system call creates an event flag specified by flgid. It dynamically allocates an event flag management control block from system memory. In addition, the initial value specified by event flag creation information, i.e. iflgptn, is set as a bit pattern for that event flag.

When the event flag generation information packet is not placed in ROM domain, i.e. when information packet is not const data type, the information definition packet is copied to the system memory.

The structure of the event flag generation information packet is as shown below.

```
Typedef struct t_cflg
{     ATR flgatr;          Event flag attribute
      FLGPTN iflgptn;      Initial value of an event flag
      B *name;             Pointer to event flag name string（Optional）
}T_CFLG;
```

TBIT_FLGPTN macro defines the number of flag bits that can be used.

Following are the valid input values for flgatr i.e. event flag attribute.

TA_WSGL	Multiple task waiting is not allowed
TA_WMUL	Multiple task waiting is allowed
TA_TFIFO	Wait task processing is in the order of arrival (FIFO)
TA_TPRI	Wait task processing is in the order of task priority
TA_CLR	Clear all flag bits at the time of task wait release

The tasks waiting in queue are not necessarily released in the order of waiting queue. The tasks are released from waiting when it matches the corresponding flag bit pattern. When TA_CLR is not specified, two or more task may be simultaneously released from waiting. When TA_CLR is specified, since the flag clears as soon as the first task is released from waiting, multiple tasks are not released simultaneously.

When TA_WSGL is specified, it is meaningless to specify TA_FIFO or TA_TPRI

Since name is an object for correspondence debugger, please specify “” or NULL as default specification. You may omit name when object structure is defined with an initial value.

Return	E_OK	Successful termination.
	E_ID	Event flag ID is outside valid range*
	E_OBJ	The event flag already exists.
	E_CTX	The command issued from an interrupt handler*
	E_SYS	Insufficient memory for the management block**

Example

```
#define ID_flg1    1
const T_CFLG cflg1 = {TA_WMUL, 0};

TASK task1(void)
{
        ER ercd;
                :
        ercd =cre_flg(ID_flg1, &cflg1);
                :
}
```

acre_flg

Function Event flag creation (automatic ID allocation)

Declaration ER_ID acre_flg(const T_CFLG *pk_cflg);

pk_cflg Pointer to event flag generation information packet

Description This system call assigns the highest value of ID searched among the non-generated event flags. When no event flag ID is allocated, the system call returns an E_NOID error. Otherwise, this is the same as cre_flg.

Return When it is non-zero positive value, the return value indicates the event flag ID.

E_NOID	Insufficient ID for Event flag.
E_CTX	The command issued from an interrupt handler*
E_SYS	Insufficient memory for the management block**

Example

```
ID ID_flg1;
const T_CFLG cflg1 = {TA_WMUL, 0};

TASK task1(void)
{
        ER_ID ercd;
                :
        ercd = acre_flg(&cflg1);
        if(ercd > 0)
        ID_flg1 = ercd;
                :
}
```

del_flg

Function Event flag deletion

Declaration ER del_flg(ID flgid);

flgid Event flag ID

Description This system call deletes an event flag specified by flgid. It releases an event flag management block back to system memory.

When a task is waiting for this event flag, the del_flg system call cancels the task waiting. An E_DLT error will be returned by the wait-cancelled task, indicating that the event flag was deleted during waiting.

Return

E_OK	Successful termination.
E_ID	Event flag ID is outside valid range*
E_NOEXS	The event flag is not generated
E_CTX	The command issued from an interrupt handler*

Example

```
#define ID_flg1    1

TASK task1(void)
{
            :
    del_flg(ID_flg1);
            :
}
```

set_flg
iset_flg

Function	Setting of event flag	
Declaration	ER set_flg(ID flgid, FLGPTN setptn);	
	ER iset_flg(ID flgid, FLGPTN setptn);	
	flgid	Event flag ID
	setptn	The bit pattern to set

Description

This system call sets up bits, indicated by setptn, for an event flag specified by flgid. In other words, a logical OR is taken with the value of setptn to the value of the present event flag (flgptn |= setptn).

As a result of changing the value of the event flag, the tasks that were waiting for the event flag are releases from waiting if the wait-release conditions are matched. This system call transfers the task from the WAITING state to the READY state (When the task has higher priority than the current running task, the system call transfers it to the RUNNING state and when in the WAITING-SUSPEND state, the set_flg system call transfers it to the SUSPENDED state).

When TA_CLR is specified during event flag creation, and if there is a task that has been released from waiting, then the event flag is cleared as soon as the first task is released from waiting.

When TA_CLR is not specified and waiting for multiple tasks is allowed, with single set_flg, multiple tasks may get released simultaneously. Depending on the relation between waiptn and wfmode in wai_flg, existence of TA_CLR in generation information, it is not necessary that the top-most task from the queue get wait released. Also, if there are tasks with clear specification waiting in the queue and these are released from waiting, then the subsequent waiting tasks lined up behind will not be released as they will refer to the cleared event flags.

Return	E_OK	Successful termination
	E_ID	Event flag ID is outside valid range*
	E_NOEXS	The event flag is not generated

Example

```
#define ID_flg1    1
#define BIT0    0x0001

TASK task1(void)
{
            :
        set_flg(ID_flg1, BIT0);
            :
}
```

clr_flg

Function Clearing of event flag

Declaration ER clr_flg(ID flgid, FLGPTN clrptn);

flgid	Event flag D
clrptn	The bit pattern to clear

Description This system call clears the bits, which are 0 by clrptn, for an event flag specified by flgid. Logical AND is taken with clrptn value and the current value of event flag.
(flgptn &= clrptn)

By clr_flg system call, the task waiting for the event flag are not released from waiting.

Return

E_OK	Successful termination.
E_ID	Event flag ID is outside valid range*
E_NOEXS	Event flag ID does not exist

Example

```
#define ID_flg1    1
#define BIT0     0x0001

TASK task1(void)
{
            :
    clr_flg(ID_flg1, ~BIT0);
            :
}
```

wai_flg

Function	Wait for event flag

Declaration ER wai_flg(ID flgid, FLGPTN waiptn, MODE wfmode, flgptn *p_flgptn);

flgid	Event flag ID
waiptn	Waiting bit pattern
wfmode	Waiting mode
p_flgptn	Pointer to the location which stores the bit pattern for wait release.

Description According to the wait conditions indicated by waiptn and wfmode, this system call waits for an event flag specified by flgid is set.

Please put the following values in wfmode to specify waiting mode.

TWF_ANDW	Waiting for AND
TWF_ORW	Waiting for OR
TWF_ANDW | TWF_CLR	Waiting for CLEAR specified AND
TWF_ORW | TWF_CLR	Waiting for CLEAR specified OR

When TWF_ORW is specified, the system call waits for either of the bits specified by waiptn to be set. When TWF_ANDW is specified, it waits for all the bits specified by waiptn to be set. When there is only one bit=1 in waiptn, TWF_ANDW and TWF_ORW have the same results.

When TWF_CLR is specified, if the conditions are satisfied and the task is released from waiting, then the wai_flg system call clears all bits for the event flag. But when TA_CLR is specified by the creation information as the flag attribute, all bits are cleared even if TWF_CLR is not specified.

An event flag value for wait release, is returned to *p_flgptn. When clearing is specified, the value before being cleared is passed to *p_flgptn. When the event flag conditions are already matched, the above operation is carried out without entering the wait state.

Return	E_OK	Successful termination.
	E_PAR	Incorrect waiting mode value in wfmode*
		Waiting bit pattern waiptn = 0*
	E_ID	Event flag ID is outside valid range*
	E_NOEXS	Event flag ID does not exist
	E_ILUSE	Waiting task already exists (when waiting for multiple tasks is not allowed)
	E_CTX	Waiting either from non-task context or in dispatch prohibited state*
	E_RLWAI	Waiting state was released forcibly (rel_wai was issued while waiting)
	E_DLT	Event flag was deleted while waiting

Note Is same as twai_flg(flgid, waiptn, wfmode, p_flgptn, TMO_FEVR).

Example

```
#define ID_flg1    1
#define BIT0      0x0001

TASK task1(void)
{
        FLGPTN ptn;
              :
       wai_flg(ID_flg1, BIT0, TWF_ANDW, &ptn);
              :
}
```

pol_flg

Function	Wait for event flag (Polling mode)	
Declaration	ER pol_flg(ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn);	
	flgid	Event flag ID
	waiptn	Waiting bit pattern
	wfmode	Waiting mode
	p_flgptn	Pointer to the location which stores the bit pattern for wait release.

Description

According to the wait conditions indicated by waiptn and wfmode, this system call waits for an event flag specified by flgid is set. The function terminates normally when the wait conditions have already been satisfied, or else function returns with E_TMOUT error value.

An event flag value for wait release, is returned to *p_flgptn. When clearing is specified, the value before being cleared is passed to *p_flgptn.

For information about wfmode, please refer to wai_flg explanation.

Return

E_OK	Successful termination.
E_PAR	Incorrect waiting mode value in wfmode*
	Waiting bit pattern waiptn = 0*
E_ID	Event flag ID is outside valid range*
E_NOEXS	Event flag ID does not exist
E_ILUSE	Waiting task already exists (when waiting for multiple tasks is not allowed)
E_TMOUT	Polling failure

Note

It is same as twai_flg(flgid, waiptn, wfmode, p_flgptn, TMO_POL)

Example

```
#define ID_flg1    1

TASK task1(void)
{
        FLGPTN ptn;
                :
        if(pol_flg(ID_flg1, 0xffff, TWF_ORW|TWF_CLR, &ptn) == E_OK)
                :
}
```

twai_flg

Function	Wait for event flag (Timeout available)

Declaration ER twai_flg(ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn, TMO tmout);

flgid	Event flag ID
waiptn	Waiting bit pattern
wfmode	Waiting mode
p_flgptn	Pointer to the location which stores the bit pattern for wait release.
Tmout	Timeout value

Description According to the wait conditions indicated by waiptn and wfmode, this system call waits for an event flag specified by flgid is set. When the wait conditions have already been satisfied, the system call does not enter the WAITING state and it terminates normally.

When the time specified by tmout passes, the call resturns with E_TMOUT time-out error. The twai_flg system call does not execute waits for tmout=TMO_POL (=0), i.e. it executes in the same way as pol_flg. For tmout=TMO_FEVR (=-1), this system call does not execute timeout, i.e. it executes the same way as wai_flg.

For information on wfmode and p_flgptn, please refer to wai_flg explanation.

Return

E_OK	Successful termination.
E_PAR	Incorrect waiting mode value in wfmode*
	Waiting bit pattern waiptn = 0*
E_ID	Event flag ID is outside valid range*
E_NOEXS	Event flag ID does not exist
E_OBJ	Waiting task already exists (when multiple waiting is not allowed)
E_CTX	Waiting either from non-task context or in dispatch prohibited state*
E_RLWAI	Waiting state was released forcibly (rel_wai was issued while waiting)
E_DLT	Event flag was deleted while waiting
E_TMOUT	Timeout

Example

```
#define ID_flg1    1

TASK task1(void)
{
        FLGPTN ptn;
        ER ercd;
                :
        ercd = twai_flg(ID_flg1, 0xffff, TWF_ANDW|TWF_CLR, &ptn, 1000/MSEC);
        if(ercd == E_TMOUT)
                :
}
```

ref_flg

Function Refer to an event flag state

Declaration ER ref_flg(ID flgid, T_RFLG *pk_rflg);

flgid	Event flag ID
pk_rflg	Pointer to a location where an event flag state packet Is stored.

Description This system call returns the state of the event flag specified by flgid to *pk_rflg.

The structure of event flag state packet is as shown below.

```
Typedef struct t_rflg
{       ID wtskid;          The waiting task ID or TSK_NONE
        FLGPTN flgptn;      The current bit pattern
}T_RFLG;
```

When the waiting task exists, the ID of the heading task in the waiting queue is returned in wtskid. When there is no waiting task, wtskid=TSK_NONE.

Return

E_OK	Successful termination.
E_ID	Event flag ID is outside valid range
E_NOEXS	Event flag ID does not exist

Example

```
#define ID_flg1    1

TASK task1(void)
{
        T_RFLG rflg;
                :
        ref_flg(ID_flg1, &rflg);
        if (rflg.flgptn != 0)
                :
}
```

5.6 Synchronization / communication functions (Data queue)

cre_dtq

Function Data queue creation

Declaration ER cre_dtq(ID dtqid, const T_CDTQ *pk_cdtq);

Dtqid	Data Queue ID
pk_cdtq	Pointer to data queue creation information packet

Description The cre_dtq system call creates a data queue specified by dtqid. The sata queue management block is dynamically allocated from system memory.

When a data queue creation information packet is placed in memory domain other than ROM (i.e. when a const data type is not attached), the creation information packet data is copied to the system memory.

Data queue creation information packet structure is as shown below.

```
typedef struct t_cdtq
{       ATR dtqatr;          Data Queue attribute
        UINT dtqcnt;         Data queue size (Byte count)
        VP dtq;              Data buffer start address
        B *name;             Data queue name string pointer (optional)
}T_CDTQ;
```

Please put following values to data queue attribute parameter, i.e. dtqatr.

TA_TFIFO	Transmission waiting queue for Data queue is in the order of arrival (FIFO)
TA_TPRI	Transmission waiting queue for Data queue is in the order of task priority

The reception-waiting queue for Data queue is always in the order of arrival (FIFO). The transmission order becomes same as the data oder. However, when forced sending (fsnd_dtq, ifsnd_dtq) is used, the forcibly sent data may be received first.

Please set the queueing data count (number of bytes) in dtqcnt, and set the data buffer start address in dtq. The size of memory required for data number n, can be found using TSZ_DTQ(n) macro. If NULL is set in dtq, the data buffer will be allocated from system memory. If 0 is set in dtqcnt, the data between tasks can be directly passed and synchronized without using the buffer.

Since name is an object for correspondence debugger, please specify “” or NULL as default specification. You may omit name when object structure is defined with an initial value.

Return	E_OK	Successful termination.
	E_ID	Data Queue ID is outside valid range*
	E_OBJ	Data Queue is already created
	E_CTX	The command issued from an interrupt handler*
	E_SYS	Insufficient system memory for management block**

Example

```
#define ID_dtq1    1

const T_CDTQ cdtq1 = {TA_TPRI, 30, NULL};

TASK task1(void)
{
        ER ercd;
                :
        ercd = cre_dtq(ID_dtq1, &cdtq1);
                :
}
```

acre_dtq

Function Data queue creation (Automatic ID allocation)

Declaration ER_ID acre_dtq(const T_CDTQ *pk_cdtq);

pk_cdtq Pointer to data queue creation information packet

Description This system call assigns the highest ID value searched among the non-generated data queue ID values. In case of failure to search the data queue ID, this system call returns with E_NOID error code. Except above differences, this system call is same as cre_dtq

Return When this call is successful, the positive return value is the allocated data queue ID.

E_NOID	Insufficient ID for Data Queue
E_CTX	The command issued from an interrupt handler*
E_SYS	Insufficient system memory for management block**

Example

```
ID ID_dtq1;
const T_CDTQ cdtq1 = {TA_TPRI, 30, NULL};

TASK task1(void)
{
    ER_ID ercd;
        :
    ercd = acre_dtq(&cdtq1);
    if(rcd > 0)
    ID_dtq1 = ercd;
        :
}
```

del_dtq

Function Delete Data queue

Declaration ER del_dtq(ID dtqid);

dtqid Data Queue ID

Description The del_dtq system call deletes a data queue specified by dtqid. The data queue management block is released to the system memory. The data buffer will also be released in case OS allocated the data buffer. The data inside the buffer is cancelled.

When any task is waiting for this data queue, this system call releases that task waiting. The released task returns with E_DLT error code indicating that the data queue was deleted while the task was waiting.

Return

E_OK	Successful termination.
E_ID	Data Queue ID is outside valid range*
E_NOEXS	Data Queue do not exist
E_CTX	The command issued from an interrupt handler*

Example

```
#define ID_dtq1    1

TASK task1(void)
{
            :
    del_dtq(ID_dtq1);
            :
}
```

snd_dtq

Function	Send Data

Declaration ER snd_dtq(ID dtqid,VP_INT data);

dtqid	Data Queue ID
data	Data to send

Description This system call sends data to a data queue specified by dtqid.

When there are tasks waiting for this data queue, this system call will release the top most waiting task in the queue, i.e. the task is changed from the WAITING state to the READY state (when the waiting task priority higher than the current running task, it is changed to the RUNNING state, and when in the WAITING-SUSPENDED state, it changes to SUSPENDED state).

When no task is waiting to receive, the data is put in the end of the data buffer. When there is no empty space in the data buffer, the task is connected to the send-waiting queue.

Return

E_OK	Successful termination.
E_ID	Data Queue ID is outside valid range*
E_NOEXS	Data Queue do not exist
E_RLWAI	Waiting state was released forcibly (rel_wai was issued while waiting)
E_DLT	Data Queue was deleted while waiting
E_CTX	Issued from the non-task context or while the dispatch is prohibited.

Note It is same as tsnd_dtq(dtqid, data, TMO_FEVR) system call.

Example

```
#define ID_dtq1    1

TASK task1(void)
{
        VP_INT data
                :
        data = (VP_INT) 1;
        snd_dtq(ID_dtq1, data);
                :
}
```

psnd_dtq
ipsnd_dtq

Function Send Data (Polling mode)

Declaration ER psnd_dtq(ID dtqid,VP_INT data);

ER ipsnd_dtq(ID dtqid,VP_INT data);

dtqid	Data Queue ID
data	Data to send

Description This system call sends data to a data queue specified by dtqid.

When there are tasks waiting for this data queue, this system call will release the top most waiting task in the queue, i.e. the task is changed from the WAITING state to the READY state (when the waiting task priority higher than the current running task, it is changed to the RUNNING state, and when in the WAITING-SUSPENDED state, it changes to SUSPENDED state).

When no task is waiting to receive, the data is put in the end of the data buffer. When there is no empty space in the data buffer, this system call immediately returns back with E_TMOUT error code. Moreover, when data buffer size is 0 and if there is no waiting task to receive data, then this call returns with E_TMOUT error.

Return

E_OK	Successful termination.
E_ID	Data Queue ID is outside valid range*
E_NOEXS	Data Queue do not exist
E_TMOUT	Polling failure

Note It is same as tsnd_dtq(dtqid, data, TMO_POL) system call.

Example

```
#define ID_dtq1    1

TASK task1(void)
{
        VP_INT data;
        ER ercd;
                :
        data = (VP_INT) 1;
        ercd = psnd_dtq(ID_dtq1, data);
        if(ercd == E_OK)
                :
                :
}
```

tsnd_dtq

Function	Send Data (Timeout available)

Declaration ER tsnd_dtq(ID dtqid, VP_INT data, TMO tmout);

dtqid	Data Queue ID
data	Data to send
tmout	Timeout value

Description This system call sends data to a data queue specified by dtqid.

When there are tasks waiting for this data queue, this system call will release the top most waiting task in the queue, i.e. the task is changed from the WAITING state to the READY state (when the waiting task priority higher than the current running task, it is changed to the RUNNING state, and when in the WAITING-SUSPENDED state, it changes to SUSPENDED state).

When no task is waiting to receive, the data is put in the end of the data buffer. When there is no empty space in the data buffer, the task is connected to the send-waiting queue.

If there is no empty space in data buffer within the time specified by tmout, this system call returns an E_TMOUT time-out error. When this system call is issued with tmout=TMO_POL (=0), the call executes similar to psnd_dtq, i.e. it does not perform waiting to send data. For tmout=TMO_FEVR (=-1), this system call runs same as snd_dtq, i.e. there is no timeout.

Return

E_OK	Successful termination.
E_ID	Data Queue ID is outside valid range*
E_NOEXS	Data Queue do not exist
E_RLWAI	Waiting state was released forcibly (rel_wai was issued while waiting)
E_DLT	Data Queue was deleted while waiting
E_CTX	Issued from the non-task context or while the dispatch is prohibited
E_TMOUT	Timeout

Example

```
#define ID_dtq1    1

TASK task1(void)
{
        VP_INT data;
        ER ercd;
                :
        data = (VP_INT) 1;
        ercd = tsnd_dtq(ID_dtq1, data, 1000/MSEC);
        if (ercd != E_TMOUT)
                :
                :
}
```

fsnd_dtq
ifsnd_dtq

Function Forced data transmission

Declaration

ER fsnd_dtq(ID dtqid, VP_INT data);
ER ifsnd_dtq(ID dtqid, VP_INT data);

dtqid	Data Queue ID
data	Data to send

Description

This system call forcibly sends data to a data queue specified by dtqid.

When there are tasks waiting for this data queue, this system call will pass the data and release the top most waiting task in the queue, i.e. the task is changed from the WAITING state to the READY state (when the waiting task priority higher than the current running task, it is changed to the RUNNING state, and when in the WAITING-SUSPENDED state, it changes to SUSPENDED state).

When no task is waiting to receive, the data is put in the end of the data buffer. When there is no empty space in the data buffer, this system call will disacrd the data that is top in the queue, and will replace that with the forced data. Data is put into the buffer even when there is other waiting task for transmission.

When data buffer size is 0 and if there is no waiting task to receive data, then this call returns with E_ILUSE error.

Return

E_OK	Successful termination.
E_ID	Data Queue ID is outside valid range*
E_NOEXS	Data Queue do not exist
E_ILUSE	Buffer size is 0

Example

```
#define ID_dtq1    1

TASK task1(void)
{
        VP_INT data;
                :
        data = (VP_INT) 1;
        fsnd_dtq(ID_dtq1, data);
                :
}
```

rcv_dtq

Function	Receive data from Data queue

Declaration ER rcv_dtq(ID dtqid, VP_INT *p_data);

dtqid	Data Queue ID
p_data	Memory pointer to location where received data is stored.

Description This system call receives a data from the first task in data queue specified by dtqid. When there are tasks waiting to send, the data to be sent is put in the data queue and the send-waiting task is released. When the data queue size is 0, data is received from the heading task in send-waiting queue. The send-waiting task is released after data reception.

When there is no data or task waiting to send, the calling task is connected to the queue of tasks waiting to receive.

Return

E_OK	Successful termination.
E_ID	Data Queue ID is outside valid range*
E_NOEXS	Data Queue do not exist
E_CTX	Waiting either from non-task context or in dispatch prohibited state*
E_RLWAI	Waiting state was released forcibly (rel_wai was issued while waiting)
E_DLT	Data Queue was deleted while waiting

Note It is same as trcv_dtq(dtqid, p_data, TMO_FEVER).

Example

```
#define ID_dtq1    1

TASK task1(void)
{
        VP_INT data;
                :
        rcv_dtq(ID_dtq1, &data);
                :
}
```

prcv_dtq

Function	Receive data from Data queue (Polling mode)

Declaration ER prcv_dtq(ID dtqid, VP_INT *p_data);

dtqid	Data Queue ID
p_data	Memory pointer to location where received data is stored.

Description This system call receives a data from the first task in data queue specified by dtqid. When there are tasks waiting to send, the data to be sent is put in the data queue and the send-waiting task is released. When the data queue size is 0, data is received from the heading task in send-waiting queue. The send-waiting task is released after data reception.

When there is no data or task waiting to send, this system call returns with E_TMOUT error code.

Return

E_OK	Successful termination.
E_ID	Data Queue ID is outside valid range*
E_NOEXS	Data Queue do not exist
E_TMOUT	Polling failure

Note It is same as trcv_dtq(dtqid, p_data, TMO_POL).

Example

```
#define ID_dtq1    1

TASK task1(void)
{
        VP_INT data;
                :
        if(prcv_dtq(ID_dtq1, &data) == E_OK)
                :
}
```

trcv_dtq

Function Receive data from Data queue (Timeout available)

Declaration ER trcv_dtq(ID dtqid, VP_INT *p_data, TMO tmout);

dtqid	Data Queue ID
p_data	Memory pointer to location where received data is stored.
tmout	Timeout value

Description This system call receives a data from the first task in data queue specified by dtqid. When there are tasks waiting to send, the data to be sent is put in the data queue and the send-waiting task is released. When the data queue size is 0, data is received from the heading task in send-waiting queue. The send-waiting task is released after data reception.

If no message is received within the time specified by tmout, this system call returns an E_TMOUT time-out error. When this system call is issued with tmout=TMO_POL (=0), the call executes similar to prcv_dtq, i.e. it does not perform waiting for data when there is no data in queue. For tmout=TMO_FEVR (=-1), this system call runs same as rcv_dtq, i.e. there is no timeout.

Return

E_OK	Successful termination.
E_ID	Data Queue ID is outside valid range*
E_NOEXS	Data Queue do not exist
E_CTX	Waiting either from non-task context or in dispatch prohibited state*
E_RLWAI	Waiting state was released forcibly (rel_wai was issued while waiting)
E_DLT	Data Queue was deleted while waiting
E_TMOUT	Timeout

Example

```
#define ID_dtq1    1

TASK task1(void)
{
        VP_INT data;
        ER ercd;
                :
        ercd = trcv_dtq(ID_dtq1, &data, 1000/MSEC);
        if(ercd == E_TMOUT)
                :
}
```

ref_dtq

Function Refer to data queue state

Declaration ER ref_dtq(ID dtqid, T_RDTQ *pk_rdtq);

dtqid	Data queue ID
pk_rdtq	Pointer to the location where data queue state packet is stored

Description This system call collects the state of the data queue specified by dtqid. The state reference information is returned in *pk_rdtq structure.

Following is the structure for data queue state packet.

```
typedef struct t_rdtq
{       ID stskid;          Task ID waiting for transmission or TSK_NONE
        ID rtskid;          Task ID waiting for reception or TSK_NONE
        UINT sdtqcnt;       Data count in data queue
}T_RDTQ;
```

When there is a waiting task, stskid & rtskidreturns the task ID number of the waiting task. TSK_NONE is returned when there is no waiting task.

Return

E_OK	Successful termination.
E_ID	Data Queue ID is outside valid range
E_NOEXS	Data Queue do not exist

Example

```
#define ID_dtq1    1

TASK task1(void)
{
        T_RDTQ rdtq;
                :
        ref_dtq(ID_dtq1, &rdtq);
        if(rdtq.sdtqcnt != 0)
                :
}
```

5.7 Synchronization / communication functions (Mail Box)

cre_mbx

Function Mailbox creation

Declaration ER cre_mbx(ID mbxid, const T_CMBX *pk_cmbx);

mbxid	Mailbox ID
pk_cmbx	Pointer to mailbox creation information packet

Description The cre_mbx system call creates a mailbox specified by mbxid. It dynamically allocates a control block for the mailbox from system memory.

Mailbox creation information packet structure is shown below.

```
typedef struct t_cmbx
{   ATR mbxatr;      Mailbox attribute
    PRI maxmpri;     Maximum message priority
    VP mprihd;       Message queue header start address
    B *name;         Pointer to the mailbox name string（optional）
}T_CMBX;
```

Please select any of following for mailbox attribute, mbxatr.

TA_TFIFO	Mailbox reception waiting task processing in the order of arrival (FIFO).
TA_TPRI	Mailbox reception waiting task processing is in the order of task priority.
TA_MFIFO	Message queuing is in the order of arrival (FIFO).
TA_MPRI	Message queuing is in the order of message priority.

When TA_MPRI is specified in mbxatr, a message queue is formed with the order of message priority. The size of the message queue header can be defined by using TSZ_MPRHD macro. When a queuing header is prepared in the user area, please ensure the memory area of number of bytes defined by TSZ_MPRHD and specify the head address as mprihd. When NULL is specified in mprihd, the queue header is allocated from the system memory.

Set the maximum value of message priority in maxmpri. Be careful in setting maxmpri, since large amount of memory is consumed for higher value of maxmpri. Similar to task priority, lower value indicates the higher message priority and the priority decreases as the value increases.

Since name is for debugger correspondence, please set “” or NULL when none is selected. You may omit name when creation information structure object is defined with initial value.

Return	E_OK	Successful termination.
	E_ID	Mailbox ID is outside valid range*
	E_OBJ	The mailbox is already generated.
	E_CTX	The command issued from an interrupt handler*
	E_SYS	Insufficient system memory for management block**

Example

```
#define ID_mbx1    1
const T_CMBX cmbx1 = {TA_TFIFO|TA_MFIFO, 1, NULL};

TASK task1(void)
{
        ER ercd;
                :
        ercd = cre_mbx(ID_mbx1, &cmbx1);
                :
}
```

acre_mbx

Function Mailbox creation (Automatic ID allocation)

Declaration ER_ID acre_mbx(const T_CMBX *pk_cmbx);

pk_cmbx	Pointer to the mailbox creation information packer

Description This system call allocates the highest ID value searched from non-generated mailbox ID values. System call will return with E_NOID error when a mailbox ID allocation fails. Except above the other part is same as cre_mbx system call.

Return A positive value indicates the allocated ID for mailbox.

E_NOID	Insufficient ID for mailbox
E_CTX	The command issued from an interrupt handler*
E_SYS	Insufficient system memory for management block**

Example

```
ID ID_mbx1;
const T_CMBX cmbx1 = {TA_TFIFO|TA_MFIFO, 1, NULL };

TASK task1(void)
{
    ER_ID ercd;
        :
    ercd = acre_mbx(&cmbx1);
    if(ercd > 0)
        ID_mbx1 = ercd;
}
```

del_mbx

Function Delete Mailbox

Declaration ER del_mbx(ID mbxid);

mbxid Mailbox ID

Description The del_mbx system call deletes a mailbox specified by mbxid. The memory allocated at the time of mailbox creation i.e.management control block etc. is released back to the system memory.

When a task is waiting for a message to be received by this mailbox, the system call releases this task from waiting. The task, whose wait was released, returns an E_DLT error indicating mailbox deletion.

A queued message if any will be lost. When the message is allocated dynamically from the memory pool, before deleting the mail box please read the message using prcv_msg and return it to a suitable memory pool. Since OS cannot automatically release all memory resources allocated by user, the memory leak may occur.

Return

E_OK	Successful termination.
E_ID	Mailbox ID is outside valid range*
E_NOEXS	The mailbox is not generated
E_CTX	The command issued from an interrupt handler*

Example

```
#define ID_mbx1    1

TASK task1(void)
{
            :
    del_mbx(ID_mbx1);
            :
}
```

snd_mbx

Function Send to Mailbox

Declaration ER snd_mbx(ID mbxid, T_MSG *pk_msg);

mbxid	Mailbox ID
pk_msg	Pointer to the message packet

Description This system call sends a message indicated by pk_msg, to the mailbox specified by mbxid. Only a pointer (value of pk_msg) is send, i.e. the contents of the message are not copied. The OS is not concerned with message size.

When no task is waiting for this mailbox, the snd_msg system call connects the message to the message queue for that mailbox and returns immediately.

When there are tasks waiting for this mailbox, the system call passes message to the top most waiting task in the queue and releases the wait. This system call transfers the task from the WAITING state to the READY state (when the waiting task priority higher than the current running task, the snd_mbx system call transfers a task to the RUNNING state, and when in the WAITING-SUSPENDED state, it changes to SUSPENDED state).

The T_MSG type structure defined as a standard message packet is shown below.

```
typedef struct t_msg
{     struct t_msg *next;        Pointer to the next message
      VB msgcont[MSGS] ;         Message contents
}T_MSG;
```

For queuing messages, the OS uses next from the message header part as a pointer. It is the part after msgcont in message header where user can actually put the message. The T_MSG type is a prototype declaration of the system call function and should not be used by the user program. As in the user program define the message structure according to use and pass to the system call with implicit casting as either (T_MSG*) or (T_MSG**). When message priority is used, set INT msgpri in addition to next in the header structure (please refer to Example2). Since the domain, which OS uses, is destroyed in case of snd_mbx, please do perform multiplex transmission.

Return

E_OK	Successful termination.
E_ID	Mailbox ID is outside valid range*
E_NOEXS	The mailbox is not generated

Note Though the standard length of message MSGS is 16bit, users can #define MSGS as a separate value before #include "kernel.h" (See Example 1).

It is better to have user defined part in the message packet structure object after msgcont, as per the actual user requirement (see Example2). msgpri definition can be omitted if the message priority order is specified as the queueing order at the time of mailbox creation. Since messages are queued without actually copying, please allocate each message a separate domain (memory pool etc). When single global variable is used, multiplex transmission problem can occur if two or more messages are queued.

Moreover, allocation of the automatic variables inside the function is prohibited to avoid erroneous operation.

Example 1

```
#define MSGS 4
#include "kernel.h"
#define ID_mbx    1
#define ID_mpf    1

TASK task1(void)
{
        T_MSG *msg;
                :
        get_mpf(ID_mpf, &msg);        /* Get the message domain */
        msg->msgcont[0] = 2;
        msg->msgcont[1] = 0;
        msg->msgcont[2] = 3;
        msg->msgcont[3] = 0;
        snd_mbx(ID_mbx, msg);         /* Send the message to mailbox */
                :
}
```

Example 2

```
typedef struct t_mymsg
{       struct t_mymsg *next;           /* The pointer to the following message (*1) */
        INT msgpri;       /* Message priority (need not be defined when not using) */
        H fncd;
        H data;
}T_MYMSG;

#define ID_mbx 1
#define ID_mpf 1

TASK task1(void)
{
        T_MYMSG *msg;
              :
        get_mpf(ID_mpf, &msg);        /* Message domain is obtained */
        msg->msgpri = 1;      /* Message priority (need not be defined when not using) */
        msg->fncd = 2;
        msg->data = 3;
        snd_mbx(ID_mbx, (T_MSG *)msg);  /* Message send to mailbox */
              :
}
```

(*1)For the system processing FAR pointer, please describe as following

```
struct t_mymsg PFAR *next;
```

rcv_mbx

Function	Mailbox reception

Declaration

ER rcv_mbx(ID mbxid, T_MSG **ppk_msg);

mbxid	Mailbox ID
ppk_msg	Pointer to the location which stores the pointer to the message packet.

Description

This system call receives a message from the mailbox specified by mbxid.The contents of messages are not copied instead only the message pointer is passed to *ppk_msg.

When messages have already been queued, the system call puts a top message pointer to ppk_msg and returns immediately. When no messages have arrived in the mailbox yet, the task issuing this system call is connected to the queue waiting for the mailbox.

Return

E_OK	Successful termination.
E_ID	Mailbox ID is outside valid range*
E_NOEXS	The mailbox is not generated
E_CTX	Waiting either from non-task context or in dispatch prohibited state*
E_RLWAI	Waiting state was released forcibly (rel_wai was issued while waiting)
E_DLT	Mailbox was deleted while waiting

Note1

ppk_msg is a double pointer.

Note2

It is same as trcv_mbx(ppk_msg, mbxid, TMO_FEVR)

In case the message sending task has acquired message domain from memory pool, the receiver side task should release the message memory to the same memory pool after message reception is finished.

Example

```
#define ID_mbx1    1
#define ID_mpf1    1

TASK task2(void)
{
        T_MYMSG *msg;
                :
        rcv_mbx(ID_mbx1, (T_MSG**)&msg);
                :
        rel_mpf(ID_mpf1, (VP)msg);          /* Message released to memory pool */
}
```

prcv_mbx

Function Mailbox reception (Polling mode)

Declaration ER prcv_mbx(ID mbxid, T_MSG **ppk_msg);

mbxid	Mailbox ID
ppk_msg	Pointer to the location which stores the pointer to the message packet.

Description This system call receives a message from the mailbox specified by mbxid.The contents of messages are not copied instead only the message pointer is passed to *ppk_msg.

When messages have already been queued, the system call puts a top message pointer to ppk_msg and returns immediately. When no messages have arrived in the mailbox yet, the call returns back with E_TMOUT error code without going into the WAITING state.

Return

E_OK	Successful termination.
E_ID	Mailbox ID is outside valid range*
E_NOEXS	The mailbox is not generated
E_TMOUT	Polling failure

Note1 ppk_msg is a double pointer.

Note2 It is same as trcv_mbx(ppk_msg, mbxid, ppk_msg, TMO_POL)

Example

```
#define ID_mbx1    1

TASK task1(void)
{
        T_MYMSG *msg;
        ER ercd;
                :
        ercd = prcv_mbx(ID_mbx1, (T_MSG**)&msg);
        if(ercd == E_OK)
                :
}
```

trcv_mbx

Function	Mailbox reception (Timeout available)

Declaration

ER trcv_mbx(ID mbxid, T_MSG **ppk_msg, TMO tmout);

mbxid	Mailbox ID
ppk_msg	Pointer to the location which stores the pointer to the message packet.
tmout	Timeout value

Description

This system call receives a message from the mailbox specified by mbxid.The contents of messages are not copied instead only the message pointer is passed to *ppk_msg.

When messages have already been queued, the system call puts a top message pointer to ppk_msg and returns immediately. When no messages have arrived in the mailbox yet, the task issuing this system call is connected to the queue waiting for the mailbox.

If no message arrives within the time specified by tmout, the trcv_msg system call returns with E_TMOUT timeout error. When this system call is issued with tmout=TMO_POL (=0), the call executes similar to prcv_mbx, i.e. it does not perform waiting for message when there is no message in queue. For tmout=TMO_FEVR (=-1), this system call runs same as rcv_mbx, i.e. there is no timeout.

Return

E_OK	Successful termination.
E_ID	Mailbox ID is outside valid range*
E_NOEXS	The mailbox is not generated
E_CTX	Waiting either from non-task context or in dispatch prohibited state*
E_RLWAI	Waiting state was released forcibly (rel_wai was issued while waiting)
E_DLT	Mailbox was deleted while waiting
E_TMOUT	Timeout error

Note

ppk_msg is a double pointer.

Example

```
#define ID_mbx1    1

TASK task1(void)
{
        T_MYMSG *msg;
        ER ercd;
                :
        ercd = trcv_mbx(ID_mbx1, (T_MSG **)&msg, 1000/MSEC);
        if(ercd == E_OK)
                :
}
```

ref_mbx

Function Refer to mailbox state

Declaration ER ref_mbx(ID mbxid, T_RMBX *pk_rmbx);

mbxid	Mailbox ID
pk_rmbx	Pointer to the location which stores the mailbox state packet

Description This system call returns the state of the mailbox specified by mbxid, to *pk_rmbx.

The structure of the mailbox state packet is as shown below.

```
typedef struct t_rmbx
{     ID wtskid;            The waiting task ID or TSK_NONE
      T_MSG *pk_msg;        Start address of the message packet at the head of message
                            queue.
}T_RMBX;
```

When there are tasks waiting in queue, tskid returns the ID number of the heading task. When there are no waiting tasks, it returns TSK_NONE.

Return

E_OK	Successful termination.
E_ID	Mailbox ID is outside valid range
E_NOEXS	The mailbox is not generated

Example

```
#define ID_mbx1    1

TASK task1(void)
{
      T_RMBX rmbx;
            :
      ref_mbx(ID_mbx1, &rmbx);
      if(rmbx.pk_msg != NULL)
            :
}
```

5.8 Extended synchronization / communication functions (Mutex)

cre_mtx

Function Mutex creation

Declaration ER cre_mtx(ID mtxid, const T_CMTX *pk_cmtx);

mtxid	Mutex ID
pk_cmtx	Pointer to mutex creation information packet

Description The cre_mtx system call creates a mutex specified by mtxid. It dynamically allocates a mutex management block from system memory.

When a mutex creation information packet is placed in memory domain other than ROM (i.e. when a const data type is not attached), the creation information packet data is copied to the system memory.

Mutex creation information packet structure is shown below.

```
typedef struct t_cmtx
{     ATR mtxatr;          Mutex attribute
      PRI ceilpri;         Mutex ceiling priority used by Priority Ceiling Protocol
      B *name;             Pointer to the mutex name string（optional）
}T_CMTX;
```

Please put any of following values to Mutex attribute parameter i.e. mtxatr.

TA_TFIFO	Waiting task processing in the order of arrival (FIFO)
TA_TPRI	Waiting task processing in the order of task priority
TA_INHERIT	Priority Inheritance Protocol is used
TA_CEILING	Priority Ceiling Protocol is used

When neither of TA_INHERIT or TA_CEILING is specified, fundamentally mutex offers the same functionality as that of binary semaphore. However in case of mutex, the task will be unlocked automatically when it terminates while it was locked.

When TA_INHERIT is specified, the current priority of the task is handled using priority inheritance protocol and priority inversion is prevented. While mutex is locked, if a high priority task waiting to lock the mutex enters the WAITING state, then the priority of the locked task becomes the same as the highest priority task waiting in the queue.
By doing this, a task with middle priority pre-empts the task that is locking mutex. It indirectly prevents the blocking of the higher priority task waiting to lock the mutex.

When TA_CEILING is specified, the current priority of the task is handled using priority ceiling protocol. In the priority ceiling protocol, ceilpri is used, which is specified in the creation information. When the task locks the mutex specified by TA_CEILING, the current priority of this task becomes the value specified by ceilpri. The priority value of the highest priority task is set in ceilpri, among the tasks, which commonly shares the mutex. Thus the same effect as priority inheritance protocol can be acquired.

Since name is for debugger correspondence, please set “” or NULL when none is selected. You may omit name when creation information structure object is defined with initial value.

Return	E_OK	Successful termination.
	E_ID	Mutex ID is outside valid range *
	E_OBJ	Mutex is already generated
	E_CTX	The command issued from an interrupt handler*
	E_SYS	Insufficient system memory for management block**

Example

```
#define ID_mtx1    1
const T_CMTX cmtx1 = {TA_INHERIT, 0};

TASK task1(void)
{
        ER ercd;
                :
        ercd = cre_mtx(ID_mtx1, &cmtx1);
                :
}
```

acre_mtx

Function Mutex creation (Automatic ID allocation)

Declaration ER_ID acre_mtx(const T_CMTX *pk_cmtx);

pk_cmtx Pointer to mutex creation information packet

Description This system call allocates the highest ID value searched from non-generated mutex ID values. System call will return with E_NOID error when a mutex ID allocation fails. Except above the other part is same as cre_mtx system call.

Return A positive value indicates the allocated ID for mutex.

E_NOID	Insufficient ID value for Mutex
E_CTX	The command issued from an interrupt handler*
E_SYS	Insufficient system memory for management block**

Example

```
ID ID_mtx1;
const T_CMTX cmtx1 = {TA_TFIFO, 0};

TASK task1(void)
{
    ER_ID ercd;
        :
    ercd = acre_mtx(&cmtx1);
    if(ercd > 0)
        ID_mtx1 = ercd;
        :
}
```

del_mtx

Function Delete Mutex

Declaration ER del_mtx(ID mtxid);

mtxid Mutex ID

Description The del_mtx system call deletes a mutex specified by mtxid. The mutex management block is released back to the system memory.

When a task is waiting for this mutex, the system call releases this task from waiting. The task, whose wait was released, returns an E_DLT error indicating that the mutex was deletion while the task was waiting for it.

Return

E_OK	Successful termination.
E_ID	Mutex ID is outside valid range *
E_NOEXS	Mutex is not created
E_CTX	The command issued from an interrupt handler*

Example

```
#define ID_mtx1    1

TASK task1(void)
{
            :
      del_mtx(ID_mtx1);
            :
}
```

unl_mtx

Function	Unlock the Mutex
Declaration	ER unl_mtx(ID mtxid); mtxid Mutex ID

Description This system call will unlock the mutex specified by mtxid.

If there are tasks waiting for this mutex, the heading task from the waiting queue is released from WAITING state. This system call transfers the task from the WAITING state to the READY state (when the waiting task priority higher than the current running task, this system call transfers a task to the RUNNING state, and when in the WAITING-SUSPENDED state, it changes to SUSPENDED state). The released task may lock the mutex again.

If there are no tasks waiting for lock, the lock is released.

It is not possible to unlock the mutex, which is not under lock by the issuing task.

Return		
	E_OK	Successful termination.
	E_ID	Mutex ID is outside valid range*
	E_NOEXS	Mutex is not created
	E_ILUSE	Specified mutex is not locked

loc_mtx

Function Lock the Mutex

Declaration ER loc_mtx(ID mtxid);

mtxid Mutex ID

Description When the mutex specified by mtxid is not locked, this system call will lock the mutex. In case the object mutex is already locked, the task calling this system call will be connected to the queue waiting to lock the mutex.

When the calling task has already locked the mutex, i.e. when you do multiple locks, this system call returns the E_ILUSE error. Moreover, E_ILUSE error is returned when a task, having higher base priority than the ceiling priority, locks the TA_CEILING specified mutex.

Return

E_OK	Successful termination.
E_ID	Mutex ID is outside valid range *
E_NOEXS	Mutex is not created
E_CTX	Waiting either from non-task context or in dispatch prohibited state*
E_RLWAI	Waiting task was released forcibly (rel_loc was issued in between)
E_DLT	Mutex was deleted while waiting for it
E_ILUSE	Multiple locking of mutex, ceiling priority violation

Note It is same as tloc_mtx(mtxid, TMO_FEVR).

Example

```
#define ID_mtx1    1

TASK task1(void)
{
            :
    loc_mtx(ID_mtx1);
            :
    unl_mtx(ID_mtx1);
            :
}
```

ploc_mtx

Function Lock the Mutex (Polling mode)

Declaration ER ploc_mtx(ID mtxid);

mtxid	Mutex ID

Description When the mutex specified by mtxid is not locked, this system call will lock the mutex. In case the object mutex is already locked, this call will return back with E_TMOUT error. Except this the other functionality is similar to loc_mtx system call.

Return

E_OK	Successful termination.
E_ID	Mutex ID is outside valid range *
E_NOEXS	Mutex is not created
E_ILUSE	Multiple locking of mutex, ceiling priority violation
E_TMOUT	Polling failure

Note It is same as tloc_mtx(mtxid, TMO_POL)

Example

```
if(ploc_mtx(ID_mtx1) == E_OK)
{
            :
    unl_mtx(ID_mtx1);
            :
}
```

tloc_mtx

Function	Lock the Mutex (Timeout available)

Declaration ER tloc_mtx(ID mtxid, TMO tmout);

mtxid	Mutex ID
tmout	Timeout value

Description When the mutex specified by mtxid is not locked, this system call will lock the mutex. In case the object mutex is already locked, the task calling this system call will be connected to the queue waiting to lock the mutex. If the mutex is not locked within the time specified by tmout, then this system call will return back with timeout error, E_TMOUT. Except above differences, other operation is same as loc_mtx system call.

When this system call is issued with tmout=TMO_POL (=0), the call executes similar to ploc_mtx, i.e. it does not perform waiting. For tmout=TMO_FEVR (=-1), this system call runs same as loc_mtx, i.e. there is no timeout.

Return

E_OK	Successful termination
E_ID	Mutex ID is outside valid range *
E_NOEXS	Mutex is not created
E_CTX	Waiting either from non-task context or in dispatch prohibited state*
E_RLWAI	Waiting task was released forcibly (rel_loc was issued in between)
E_DLT	Mutex was deleted while waiting for it
E_ILUSE	Multiple locking of mutex, ceiling priority violation
E_TMOUT	Timeout error

Example

```
#define ID_mtx1    1

TASK task1(void)
{
        ER ercd;
                :
        ercd = tloc_mtx(ID_mtx1, 100/MSEC);
        if(ercd == E_OK)
                :
}
```

ref_mtx

Function Refer to Mutex state

Declaration ER ref_mtx(ID mtxid, T_RMTX *pk_rmtx);

mtxid	Mutex ID
pk_rmtx	Pointer to the location where mutex state packet is stored

Description This system call returns the state of the mutex specified by mbxid, to *pk_rmtx.

Following is the structure for mutex state packet.

```
typedef struct t_rmtx
{       ID htskid;          ID of the locked task or TSK_NONE
        ID wtskid;          ID of task waiting for lock or TSK_NONE
}T_RMTX;
```

When there is a task, which had locked the object mutex, then that task ID value will be returned in htskid. TSK_NONE will be returned when there is no such task.

ID number of the heading task in the mutex queue will be returned in wtskid. When there is no waiting task, TSK_NONE is returned.

Return

E_OK	Successful termination.
E_ID	Mutex ID is outside valid range
E_NOEXS	Mutex is not created

Example

```
#define ID_mtx1    1

TASK task1(void)
{
        T_RMTX rmtx;
                :
        ref_mtx(ID_mtx1, &rmtx);
                :
}
```

5.9 Extended synchronization / communication functions (Message buffer)

cre_mbf

Function Message Buffer creation

Declaration ER cre_mbf(ID mbfid, const T_CMBF *pk_cmbf);

mbfid Message Buffer ID

pk_cmbf Pointer to message buffer creation information packet

Description The cre_mbf system call creates a message buffer specified by mbfid. Message buffer management block is dynamically allocated from system memory.

When a message buffer creation information packet is placed in memory domain other than ROM (i.e. when a const data type is not attached), the creation information packet data is copied to the system memory.

Message Buffer creation information packet structure is shown below.

```
typedef struct t_cmbf
{     ATR mbfatr;          Message buffer attribute
      UINT maxmsz;         Maximum size of message (Byte count)
      SIZE mbfsz;          Total size of ring buffer (Byte count)
      VP mbf;              Start address of ring buffer
      B *name;             Pointer to the message buffer name string（optional）
}T_CMBF;
```

Please put any of following values to message buffer attribute parameter i.e. mbfatr.

TA_TFIFO Processing of send-waiting task in the order of arrival (FIFO)

TA_TPRI Processing of send-waiting task in the order of task priority

TA_TPRIR Processing of receive-waiting task in the order of task priority

When TA_TPRIR is not specified to mbfatr, the processing of receive-waiting task is in the order of arrival.

When the ring buffer domain is secured by user program, please set the start address of ring buffer to mbf. In this case, since the part of buffer will be used for message management, all ring buffer domain cannot be used by user program.

The total size in order to store msgcnt number of messages of size msgsz bytes (msgsz > 1), can be obtained using TSZ_MBF(msgcnt, msgsz) macro definition. However, when the message size is 1 byte (msgsz=1), the memory domain of size msgsz bytes is essential i.e. thre is no overhead by OS.

When the mbf is NULL, the ring buffer memory of size defined by mbufsz, is dynamically allocated from memory pool domain.

It is also possible to set mbfsz=0. In such a case, the ring buffer is not required. When mbfsz=0, the tasks are synchronized to transfer the data directly.

Since name is for debugger correspondence, please set “” or NULL when none is selected. You may omit name when creation information structure object is defined with initial value.

Return	E_OK	Successful termination.
	E_ID	Message buffer ID is outside valid range*
	E_OBJ	Message buffer is already created
	E_PAR	Parameter error (maxmsz = 0)*
	E_CTX	The command issued from an interrupt handler*
	E_SYS	Insufficient system memory for management block**
	E_NOMEM	Insufficient memory for Ring Buffer**

Example

```
#define ID_mbf1    1
const T_CMBF cmbf1 = {TA_TFIFO, 32, 512, NULL};

TASK task1(void)
{
        ER ercd;
                :
        ercd = cre_mbf(ID_mbf1, &cmbf1);
                :
}
```

acre_mbf

Function Message Buffer creation (Automatic ID allocation)

Declaration ER_ID acre_mbf(const T_CMBF *pk_cmbf);

pk_cmbf	Pointer to message buffer creation information packet

Description This system call allocates the highest ID value searched from non-generated message buffer ID values. System call will return with E_NOID error when the ID allocation fails. Except above the other part is same as cre_mbf system call.

Return A positive value indicates the allocated ID for message buffer.

E_NOID	Insufficient ID for message buffer
E_PAR	Parameter error (maxmsz = 0)*
E_CTX	The command issued from an interrupt handler*
E_SYS	Insufficient system memory for management block**
E_NOMEM	Insufficient memory for Ring Buffer**

Example

```
ID ID_mbf1;
const T_CMBF cmbf1 = {TA_TFIFO, 32 ,512, NULL};

TASK task1(void)
{
        ER_ID ercd;
                :
        ercd = acre_mbf(&cmbf1);
        if(ercd > 0)
                ID_mbf1 = ercd;
}
```

del_mbf

Function Delete Message Buffer

Declaration ER del_mbf(ID mbfid);

mbfid Message Buffer ID

Description The del_mbf system call deletes a message buffer specified by mbfid. The message buffer management block is released to the system memory. The ring buffer domain will also be released in case OS allocated the ring buffer.

When a task is waiting this message buffer for transmission or reception, the system call releases this task from waiting. The task, whose wait was released, returns an E_DLT error indicating that the message buffer was deletion while the task was waiting for it.

Return

E_OK	Successful termination.
E_ID	Message buffer ID is outside valid range*
E_NOEXS	This message buffer is not created
E_CTX	The command issued from an interrupt handler*

Example

```
#define ID_mbf1    1

TASK task1(void)
{
            :
     del_mbf(ID_mbf1);
            :
}
```

snd_mbf

Function	Send message to Message Buffer

Declaration ER snd_mbf(ID mbfid, VP msg, UINT msgsz);

mbfid	Message buffer ID
msg	Pointer to the message to be send
msgsz	Size of transmitting message (Byte count)

Description This system call sends the message defined by msg & msgsz, to the message buffer specified by mbfid.

When there is a task waiting for the message from this message buffer, the snd_mbf system call copies the message to the receiving buffer of the heading task in the receive-waiting queue, and then releases that task from waiting.
When no tasks are waiting for the message to be received from this message buffer, this system calls copies the message to the ring buffer used by that message buffer. However, if the ring buffer is full, the task that issued this system call enters the WAITING state and waits for the message to be sent.

In order to perform queuing of messages of size msgsz in snd_mbf, psnd_mbf & tsnd_mbf system calls, the ring buffer should have the minimum free space of size,
= msgsz + 2Bytes (The header part which shows message size).
However, when message maximum length, maxmsg, is specified as 1 byte, additional 2 byte header is unnecessary.

Return

E_OK	Successful termination.
E_PAR	Message size is out of valid range (msgsz = 0 , msgsz > maxmsz of creation information)*
E_ID	Message buffer ID is outside valid range*
E_NOEXS	This message buffer is not created
E_CTX	Issued from the non-task context, or waiting in dispatch prohibited state*
E_RLWAI	Waiting state was forcibly released
E_DLT	Message buffer was deleted while waiting for it

Note It is same as tsnd_mbf(mbfid, msg, msgsz, TMO_FEVR).

Example

```
#define ID_mbf1    1

TASK task1(void)
{
        H cmd = 0x0012;
                :
        snd_mbf(ID_mbf1, (VP)&cmd, sizeof cmd);
                :
}
```

psnd_mbf

Function Send message to Message Buffer (Polling mode)

Declaration ER psnd_mbf(ID mbfid, VP msg, UINT msgsz);

mbfid	Message Buffer ID
msg	Pointer to the message to be send
msgsz	Size of transmitting message (Byte count)

Description This system call sends the message defined by msg & msgsz, to the message buffer specified by mbfid.

When there is a task waiting for the message from this message buffer, the snd_mbf system call copies the message to the receiving buffer of the heading task in the receive-waiting queue, and then releases that task from waiting.

When no tasks are waiting for the message to be received from this message buffer, this system calls copies the message to the ring buffer used by that message buffer. However, if the ring buffer is full, without entering the WAITING state, the call returns back with E_TMOUT error.

Return

E_OK	Successful termination.
E_PAR	Message size is out of valid range (msgsz = 0 , msgsz > maxmsz of creation information)*
E_ID	Message buffer ID is outside valid range*
E_NOEXS	This message buffer is not created
E_TMOUT	Polling failure

Note It is same as tsnd_mbf(mbfid, msg, msgsz, TMO_POL).

Example

```
#define ID_mbf2    2

TASK task1(void)
{
        B msg[16] ;
                :
        strcpy(msg, "Hello");
        if(psnd_mbf(ID_mbf2, (VP)msg, strlen(msg)) != E_OK)
                :
}
```

tsnd_mbf

Function	Send message to Message Buffer (Timeout available)	
Declaration	ER tsnd_mbf(ID mbfid, VP msg, UINT msgsz, TMO tmout);	
	mbfid	Message Buffer ID
	msg	Pointer to the message to sent
	msgsz	Size of transmitting message (Byte count)
	tmout	Timeout value

Description

This system call sends the message defined by msg & msgsz, to the message buffer specified by mbfid.

When there is a task waiting for the message from this message buffer, the snd_mbf system call copies the message to the receiving buffer of the heading task in the receive-waiting queue, and then releases that task from waiting.
When no tasks are waiting for the message to be received from this message buffer, this system calls copies the message to the ring buffer used by that message buffer. However, if the ring buffer is full, the task that issued this system call enters the WAITING state and waits for the message to be sent.

If there is no free space even after the time specified by tmout is passed, then this system call will return back with timeout error, E_TMOUT.

When this system call is issued with tmout=TMO_POL (=0), the call executes similar to psnd_mbf, i.e. it does not perform waiting. For tmout=TMO_FEVR (=-1), this system call runs same as snd_mbf, i.e. there is no timeout.

Return	E_OK	Successful termination.
	E_PAR	Message size is out of valid range (msgsz = 0 , msgsz > maxmsz of creation information)*
	E_ID	Message buffer ID is outside valid range*
	E_NOEXS	This message buffer is not created
	E_CTX	Issued from the non-task context, or waiting in dispatch prohibited state*
	E_RLWAI	Waiting state was released forcibly (rel_wai was issued while waiting)
	E_DLT	Message buffer was deleted while waiting for it
	E_TMOUT	Timeout

Example

```
#define ID_mbf2    2

TASK task1(void)
{
        B *res = "Hello";
        ER ercd;
                :
        ercd = tsnd_mbf(ID_mbf2, (VP)res, 5, 1000/MSEC);
        if(ercd = E_TMOUT)
                :
}
```

rcv_mbf

Function Receive message from Message Buffer

Declaration ER_UINT = rcv_mbf(ID mbfid, VP msg);

mbfid	Message Buffer ID
msg	Pointer to the location to store received message

Description The rcv_mbf system call receives a message, from the message buffer specified by mbfid. The received message is copied to msg. This system call returns the size of the received message.

The size of domain pointed by msg, need to be larger than the maximum length of message (maxmsz), which was specified t the time of message buffer creation.

When message has not arrived in the message buffer, the task that has issued this system call is connected to the queue of tasks waiting for the message to be received from this message buffer.

Return When positive, this return value indicates received message byte count.

E_ID	Message buffer ID is outside valid range*
E_NOEXS	This message buffer is not created
E_CTX	Issued from the non-task context, or waiting in dispatch prohibited state*
E_RLWAI	Waiting state was released forcibly (rel_wai was issued while waiting)
E_DLT	Message buffer was deleted while waiting for it

Note It is same as trcv_mbf(mbfid, msg, TMO_FEVR)

Example

```
#define ID_mbf1    1

TASK task1(void)
{
        H cmd;
        ER dummy;
                :
        dummy = rcv_mbf(ID_mbf1, (VP)&cmd);
                :
}
```

prcv_mbf

Function Receive message from Message Buffer (Polling mode)

Declaration ER_UINT = prcv_mbf(ID mbfid, VP msg);

mbfid	Message Buffer ID
msg	Pointer to the location to store received message

Description This system call receives a message, from the message buffer specified by mbfid. The received message is copied to msg. This system call returns the size of the received message.

The size of domain pointed by msg, need to be larger than the maximum length of message (maxmsz), which was specified t the time of message buffer creation.

When message has not arrived in the message buffer, without entering the WAITING state, this system call returns with E_TMOUT timeout error.

Return When positive, this return value indicates received message byte count.

E_ID	Message buffer ID is outside valid range*
E_NOEXS	This message buffer is not created
E_TMOUT	Polling failure

Note It is same as trcv_mbf(msg, p_msgsz, mbfid, TMO_POL).

Example

```
#define ID_mbf2    2

TASK task1(void)
{
        B buf[16] ;
        ER size;
                :
        if(size = prcv_mbf(ID_mbf2, (VP)buf) > 0)
                :
}
```

trcv_mbf

Function	Receive message from Message Buffer (Timeout available)	
Declaration	ER_UINT = trcv_mbf(ID mbfid, VP msg, TMO tmout);	
	mbfid	Message Buffer ID
	msg	Pointer to the location to store received message
	tmout	Timeout value

Description

This system call receives a message, from the message buffer specified by mbfid. The received message is copied to msg. This system call returns the size of the received message. The size of domain pointed by msg, need to be larger than the maximum length of message (maxmsz), which was specified t the time of message buffer creation.

When message has not arrived in the message buffer, the task that has issued this system call is connected to the queue of tasks waiting for the message to be received from this message buffer.

If the message is not received within the time specified by tmout, then this system call will return back with timeout error, E_TMOUT.

When this system call is issued with tmout=TMO_POL (=0), the call executes similar to prcv_mbf, i.e. it does not perform waiting. For tmout=TMO_FEVR (=-1), this system call runs same as rcv_mbf, i.e. there is no timeout.

Return

When positive, this return value indicates received message byte count.

E_ID	Message buffer ID is outside valid range*
E_NOEXS	This message buffer is not created
E_CTX	Issued from the non-task context, or waiting in dispatch prohibited state*
E_RLWAI	Waiting state was released forcibly (rel_wai was issued while waiting)
E_DLT	Message buffer was deleted while waiting for it
E_TMOUT	Timeout error

Example

```
#define ID_mbf2    2

TASK task1(void)
{
        B buf[16] ;
        ER_UINT size;
                :
        size = trcv_mbf(ID_mbf2, (VP)buf, 1000/MSEC)
        if (ercd == E_TMOUT)
                :
}
```

ref_mbf

Function Refer to state of Message Buffer

Declaration ER ref_mbf(ID mbfid, T_RMBF *pk_rmbf);

mbfid	Message Buffer ID
pk_rmbf	Pointer to the location where message buffer state packet is stored

Description This system call returns the state of the message buffer specified by mbfid, to *pk_rmbf.

Message buffer state packet structure is as shown below.

```
typedef struct t_rmbf
{       ID stskid;          ID of task waiting for transmission or TSK_NONE
        ID rtskid;          ID of task waiting for reception or TSK_NONE
        UINT smsgcnt;       The number of messages in the message buffer
        SIZE fmbfsz;        Free size in ring buffer (Byte count)
}T_RMBF;
```

ID number of the heading task in the message buffer queue will be returned in stskid and rtskid. When there is no waiting task, TSK_NONE is returned.

Return

E_OK	Successful termination.
E_ID	Message buffer ID is outside valid range
E_NOEXS	This message buffer is not created

Example

```
#define ID_mbf1    1

TASK task1(void)
{
        T_RMBF rmbf;
                :
        ref_mbf(ID_mbf1, &rmbf);
        if (rmbf.fmbufsz >= 32 + sizeof(int))
                :
}
```

5.10 Extended synchronization / communication functions (Rendezvous port)

cre_por

Function Create rendezvous port

Declaration ER cre_por(ID porid, const T_CPOR *pk_cpor);

porid	Rendezvous port ID
pk_cpor	Pointer to rendezvous port creation information packet

Description The cre_por system call creates a rendezvous port specified by porid. Port management block is dynamically allocated from system memory. When a message buffer creation information packet is placed in memory domain other than ROM (i.e. when a const data type is not attached), the creation information packet data is copied to the system memory.

Rendezvous port creation information packet structure is shown below.

```
typedef struct t_cpor
{       ATR poratr;             Rendezvous port attribute
        UINT maxcmsz;       Maximum length of calling message (Byte count)
        UINT maxrmsz;       Maximum length of return message (Byte count)
        B *name;                Port name string pointer (optional)
}T_CPOR;
```

Please set any of following values for rendezvous port attribute, i.e. poratr.

TA_TFIFO	Processing of call-waiting task in the order of arrival (FIFO)
TA_TPRI	Processing of call-waiting task in the order of task priority

Rendezvous acceptance queuing is always in the FIFO order. Since the message is copied when both calling side and the accepting side meet, in case of rendezvous, there is no ring buffer to perform queuing of messages etc.

It is also possible to set 0 values to maxcmsz and maxrmsz.

Since name is for debugger correspondence, please set “” or NULL when none is selected. You may omit name when creation information structure object is defined with initial value.

Return

E_OK	Successful termination.
E_ID	Rendezvous port ID is outside valid range*
E_OBJ	Rendezvous port is already created
E_CTX	The command issued from an interrupt handler*
E_SYS	Insufficient system memory for management block**

Example

```
#define ID_por1    1
const T_CPOR cpor1 = {TA_TFIFO, 64, 32};

TASK task1(void)
{
        ER ercd;
                :
        ercd = cre_por(ID_por1, &cpor1);
                :
}
```

acre_por

Function Create rendezvous port (Automatic ID allocation)

Declaration ER_ID acre_por(const T_CPOR *pk_cpor);

pk_cpor Pointer to rendezvous port creation information packet

Description This system call allocates the highest ID value searched from non-generated rendezvous port ID values. System call will return with E_NOID error when the ID allocation fails. Except above the other part is same as cre_por system call.

Return A positive value indicates the allocated ID for rendezvous port.

E_NOID	Insufficient ID for rendezvous port
E_CTX	The command issued from an interrupt handler*
E_SYS	Insufficient system memory for management block**

Example

```
ID ID_por1;
const T_CPOR cpor1 = {TA_TFIFO, 64, 32 };

TASK task1(void)
{
        ER_ID ercd;
                :
        ercd = acre_por(&cpor1);
        if(ercd > 0)
                ID_por1 = ercd;
                :
}
```

del_por

Function Delete rendezvous port

Declaration ER del_por(ID porid);

porid Rendezvous port ID

Description The del_por system call deletes a rendezvous port specified by porid. The rendezvous port management block is released to the system memory.

When a task is waiting this rendezvous port for rendezvous call or rendezvous acceptance, the system call releases this task from waiting. The task, whose wait was released, returns an E_DLT error indicating that the rendezvous port was deletion while the task was waiting for it.

When the rendezvous port is deleted, it does not affect the already created rendezvous ports.

Return

E_OK	Successful termination.
E_ID	Rendezvous port ID is outside valid range*
E_NOEXS	Rendezvous port is not created
E_CTX	The command issued from an interrupt handler*

Example

```
#define ID_por1    1

TASK task1(void)
{
            :
    del_por(ID_por1);
            :
}
```

cal_por

Function Call rendezvous

Declaration ER_UINT cal_por(ID porid, RDVPTN calptn, VP msg, UINT cmsgsz);

porid	Rendezvous port ID
calptn	Bit pattern of the rendezvous condition at the calling side
msg	Pointer to the calling message, and the pointer to the reply message
cmsgsz	Size of the calling message (Byte count)

Description After waiting for the accepting task, the cal_por system call will send the calling message to the accepting task, using the rendezvous port specified by porid. Furthermore, this system call will wait until the reply message is received from the accepting side task.

It is possible to select combination of calling-side and accepting side, using calptn bit pattern. Rendezvous is established when the logical AND of, calptn of cal_por (this system call issued from rendezvous calling task), and acpptn of the acp_por (system call issued by accepting task), is a non-zero value.

When there is a task waiting for accepting rendezvous at this rendezvous port, this system call will check if the rendezvous can be established with this accepting-waiting task. When there are several tasks waiting for accepting rendezvous, this system call will check for the possibility of rendezvous formation, one by one starting from the heading task in the accept-waiting queue. When there is no waiting task for the rendezvous acceptance, or when it is not possible to establish rendezvous with any of the task from accept-waiting queue, then the calling side task which has issued this system call, is connected to the rendezvous call waiting queue.

When the rendezvous is established, a calling message is copied to the buffer of accepting task, and the accepting task is released from the waiting. The calling side task which publishes this system call will enter the WAITING state, waiting for the rendezvous end. Since the task is separated from the port, there is no queue formation while waiting for rendezvous end.

The rendezvous is terminated when the reply message is received from the accepting task (by using rpl_rdv system call). The reply message is copied to msg buffer.

When successful, this system call returns the size of the reply message as return value.

The memory area pointed by msg should be larger than the maxrmsz i.e. maximum length of reply message specified at the time of rendezvous port creation.

Return When 0 or positive, the return value indicates the reply message size.

E_PAR	Bit pattern of the rendezvous condition at the calling side, calptn is 0*
	Message size is outside valid range(cmsgsz = 0, cmsgsz > maxcmsz)*
E_ID	Rendezvous port ID is outside valid range*
E_NOEXS	Rendezvous port is not created
E_CTX	Issued from the non-task context, or waiting in dispatch prohibited state*
E_RLWAI	Waiting state was released forcibly (rel_wai was issued while waiting)
E_DLT	The rendezvous port was deleted while waiting for it

Note This call is same as tcal_por(porid, calptn, msg, cmsgsz, TMO_FEVR).

Example

```
#define ID_por1    1

TASK task1(void)
{
        B msg[16] ;
        ER_UINT size;
                :
        strcpy(msg, "Hello");
        size = cal_por(ID_por1, 0x0001, (VP)msg, strlen(msg));
        if(size >= 0)
                :
}
```

tcal_por

Function Call rendezvous (Timeout available)

Declaration ER_UINT = tcal_por(ID porid, RDVPTN calptn, VP msg, UINT cmsgsz, TMO tmout);

porid	Rendezvous port ID
calptn	Bit pattern of the rendezvous condition at the calling side
msg	Pointer to the location which stores the reply message
cmsgsz	Size of the calling message (Byte count)
tmout	Timeout value

Description Following are the differences from cal_por.

If the rendezvous has not been terminated within the time specified by tmout, this system call will return back with E_TMOUT error code.

When this system call is issued with tmout=TMO_POL (=0), i.e. with polling specification, the call returns with E_PAR eror code. For tmout=TMO_FEVR (=-1), this system call runs same as cal_por, i.e. there is no timeout.

Return When 0 or positive, the return value indicates the reply message size.

E_PAR	Bit pattern of the rendezvous condition at the calling side, calptn is 0* Message size is outside valid range(cmsgsz = 0, cmsgsz > maxcmsz)* Polling mode specified*
E_ID	Rendezvous port ID is outside valid range*
E_NOEXS	Rendezvous port is not created
E_CTX	Issued from the non-task context, or waiting in dispatch prohibited state*
E_RLWAI	Waiting state was released forcibly (rel_wai was issued while waiting)
E_DLT	The rendezvous port was deleted while waiting for it
E_TMOUT	Timeout error

acp_por

Function Accept rendezvous

Declaration ER_UINT = acp_por(ID porid, RDVPTN acpptn, RDVNO *p_rdvno, VP msg);

porid	Rendezvous port ID
acpptn	Bit pattern of the rendezvous condition at the accepting side
p_rdvno	Pointer to the location where rendezvous number is stored
msg	Pointer to the calling message

Description When the waiting for the calling-task is over, the acp_por system call will accept the calling message from the calling task, using the rendezvous port specified by porid.

It is possible to select combination of calling-side and accepting side, using acpptn bit pattern. Rendezvous is established when the logical AND of, calptn of cal_por (this system call issued from rendezvous calling task), and acpptn of the acp_por (system call issued by accepting task), is a non-zero value.

When there is a task waiting for rendezvous call at this rendezvous port, this system call will check if the rendezvous can be established with the call-waiting task. When there are several tasks waiting for rendezvous call, this system call will check for the possibility of rendezvous formation, one by one starting from the heading task in the call-waiting queue. When there is no waiting task for the rendezvous call, or when it is not possible to establish rendezvous with any of the task from call-waiting queue, then the accepting side task which has issued this system call, is connected to the rendezvous accept-waiting queue.

When the rendezvous is established, the calling message is received and copied to msg. The calling side task is changed to rendezvous end-waiting state from the call-waiting state. After successful operation, this system call returns with calling-message size as the function return value.

The memory area pointed by msg should be larger than the maximum length of the calling message that was specified at the time of rendezvous port creation.

The rendezvous number, which can be used for fwd_por or rpl_por system calls later, is returned in *p_rdvno. Calling-side task while waiting for rendezvous end is detached from port. Hence instead of port number, the rendezvous number associated to the task (i.e. *p_rdvno) need to be specified for fwd_por or rpl_por system calls.

Return	When positive, the return value indicates the size of the calling message (Byte count)	
	E_PAR	Bit pattern of the rendezvous condition at the accepting side, acpptn is 0*
	E_ID	Rendezvous port ID is outside valid range*
	E_NOEXS	Rendezvous port is not created
	E_CTX	Issued from the non-task context, or waiting in dispatch prohibited state*
	E_RLWAI	Waiting state was released forcibly (rel_wai was issued while waiting)
	E_DLT	The rendezvous port was deleted while waiting for it

Note This call is same as tacp_por(porid, acpptn, p_rdvno, msg, TMO_FEVR).

Example

```
#define ID_por1    1
#define ID_por2    2

TASK task1(void)
{
        B msg[64] ;
        ER_UINT size;
        RDVNO rdvno;
                :
        strcpy(msg, "Welcome");
        size = acp_por(ID_por1, 0xffff, &rdvno, (VP)msg);
        if(memcmp(msg, "Hello", size) == 0)
        {       strcpy(msg, "World");
                rpl_rdv(rdvno, msg,strlen(msg));
        }else
                fwd_por(ID_por2, 0x0001, rdvno, msg, strlen(msg));
                :
                :
}
```

pacp_por

Function	Accept rendezvous (Polling mode)

Declaration ER_UINT = pacp_por(ID porid, RDVPTN acpptn, RDVNO *p_rdvno, VP msg);

porid	Rendezvous port ID
acpptn	Bit pattern of the rendezvous condition at the accepting side
p_rdvno	Pointer to the location where rendezvous number is stored
msg	Pointer to the calling message

Description Following are the differences from acp_por.

When there is no waiting task for rendezvous call and when rendezvous is not established at the calling task, then instead of waiting in queue, this system call returns back with E_TMOUT error.

Return When positive, the return value indicates the size of the calling message (Byte count)

E_PAR	Bit pattern of the rendezvous condition at the accepting side, acpptn is 0*
E_ID	Rendezvous port ID is outside valid range*
E_NOEXS	Rendezvous port is not created
E_TMOUT	Polling failure

Note This call is same as tacp_por(porid, acpptn, p_rdvno, msg, TMO_POL).

tacp_por

Function	Accept rendezvous (Timeout available)

Declaration ER_UINT = tacp_por(ID porid, RDVPTN acpptn, RDVNO *p_rdvno, VP msg, TMO tmout);

porid	Rendezvous port ID
acpptn	Bit pattern of the rendezvous condition at the accepting side
p_rdvno	Pointer to the location where rendezvous number is stored
msg	Pointer to the calling message
tmout	Timeout value

Description Following are the differences from acp_por.

When rendezvous is not established within the time specified by tmout, then this system call returns back with E_TMOUT error.

When this system call is issued with tmout=TMO_POL (=0), the call executes similar to pacp_por, i.e. it does not perform waiting. For tmout=TMO_FEVR (=-1), this system call runs same as acp_por, i.e. there is no timeout.

Return When positive, the return value indicates the size of the calling message (Byte count)

E_PAR	Bit pattern of the rendezvous condition at the accepting side, acpptn is 0*
E_ID	Rendezvous port ID is outside valid range*
E_NOEXS	Rendezvous port is not created
E_CTX	Issued from the non-task context, or waiting in dispatch prohibited state*
E_RLWAI	Waiting state was released forcibly (rel_wai was issued while waiting)
E_DLT	The rendezvous port was deleted while waiting for it
E_TMOUT	Timeout error

fwd_por

Function Forward rendezvous port

Declaration ER fwd_por(ID porid, RDVPTN calptn, RDVNO rdvno, VP msg, UINT cmsgsz);

porid	ID of the Rendezvous port to which the rendezvous port is forwarded
calptn	Bit pattern of the rendezvous condition at the calling side
rdvno	Rendezvous number to be forwarded
msg	Pointer to the calling message
cmsgsz	Size of the calling message (Byte count)

Description The fwd_por system call forwards the rendezvous specified by rdvno, to other port specified by porid (it is possible to specify self port ID), and allows other tasks to re-execute the rendezvous acceptance.

The calling size task which was in the rendezvous end-waiting state can be made to process the rendezvous call again from the port different than the port used last time for calling. Moreover, the bit pattern used for the rendezvous formation is replaced with the calptn bit pattern of this system call.

Using the port after forwarding, if there is a task waiting for the rendezvous acceptance, this system call will check if the rendezvous can be established with the accepting-waiting task. When there are several tasks waiting for rendezvous call, this system call will check for the possibility of rendezvous formation, one by one starting from the heading task in the accepting-waiting queue. When there is no waiting task for the rendezvous acceptance, or when it is not possible to establish rendezvous with any of the task from accepting-waiting queue, then the calling side task which is object for this system call, is connected to the rendezvous calling-waiting queue.

When the rendezvous is established, the calling message is copied to the buffer of the accepting-task. The accepting task is released from the accept-waiting state. The calling-side task, which is object for forwarding the port, will again enter the rendezvous end waiting state.

The task that issued this system call will not enter the WAITING state. This system call can be published only after rendezvous acceptance. It is possible to further forward the rendezvous to different port.

Return	E_OK	Successful termination.
	E_PAR	Bit pattern of the rendezvous condition at the calling side, calptn is 0*
		Message size is outside valid range(cmsgsz = 0, cmsgsz > maxcmsz)*
	E_ID	Rendezvous port ID is outside valid range*
	E_OBJ	Object task is not waiting for rendezvous end, or maxrmsz of the port after forwarding is larger than the maxrmsz before forwarding*
	E_NOEXS	Rendezvous port is not created

rpl_rdv

Function	Reply rendezvous

Declaration ER rpl_rdv(RDVNO rdvno, VP msg, UINT rmsgsz);

rdvno	Rendezvous number
msg	Pointer to the reply message
rmsgsz	Size of reply message (Byte count)

Description The rpl_rdv system call will send the reply message to the calling task for rendezvous specified by rdvno. Moreover, this system call will terminate the rendezvous specified by rdvno.

This system call transfers the rendezvous calling side task from the WAITING state to the READY state (when the waiting task priority higher than the current running task, the snd_mbx system call transfers a task to the RUNNING state, and when in the WAITING-SUSPENDED state, it changes to SUSPENDED state). The task which issued this system call will not enter the WAITING state.

This system call can be issued only after the acceptance of rendezvous.

Return

E_OK	Successful termination
E_PAR	Message size is outside valid range
E_OBJ	Object task is not waiting for rendezvous end*

ref_por

Function Refer to rendezvous port state

Declaration ER ref_por(ID porid, T_RPOR *pk_rpor);

porid	Rendezvous port ID
pk_rpor	Pointer to the location where rendezvous port state packet is stored

Description This system call returns the state of the rendezvous port specified by porid, to *pk_rpor.

Rendezvous port state packet structure is as shown below.

```
typedef struct t_rpor
{       ID ctskid;          ID of a task waiting for call, or TSK_NONE
        ID atskid;          ID of a task waiting for acceptance, or TSK_NONE
}T_RPOR;
```

When there is a task waiting for rendezvous port call or acceptance, then that task ID value will be returned in ctskid and atskid. TSK_NONE will be returned when there is no waiting task.

Return

E_OK	Successful termination.
E_ID	Rendezvous port ID is outside valid range
E_NOEXS	Rendezvous port is not created

Example

```
#define ID_por1    1

TASK task1(void)
{
        T_RPOR rpor;
                :
        ref_por(ID_por1, &rpor);
        if(rpor.atskid != TSK_NONE)
                :
}
```

ref_rdv

Function Refer to rendezvous state

Declaration ER ref_rdv(RDVNO rdvno, T_RRDV *pk_rrdv);

rdvno	Rendezvous number
pk_rrdv	Pointer to the location where rendezvous state packet is stored

Description This system call returns the state of the rendezvous specified by rdvno, to *pk_rrdv.

Rendezvous state packet structure is as shown below.

```
typedef struct t_rrdv
{
    ID wtskid;        ID of a task waiting for rendezvous end, or TSK_NONE
}T_RRDV;
```

When there is a task waiting for rendezvous, then that task ID value will be returned in wtskid. TSK_NONE will be returned when there is no waiting task.

Return

E_OK	Successful termination.
E_ID	Rendezvous port ID is outside valid range
E_NOEXS	Rendezvous port is not created

Example

```
TASK task1(void)
{
    T_RRDV rrdv;
    RDVNO rdvno;
        :
    ref_rdv(rdvno, &rrdv);
    if (rrdv.wtskid != TSK_NONE)
        :
}
```

5.11 Interrupt management functions

def_inh

Function Interrupt handler definition

Declaration ER def_inh(INHNO inhno, const T_DINH *pk_dinh);

inhno	Interrupt handler number
pk_dinh	Pointer to the interrupt handler definition information packet.

Description This system call will set the interrupt handler specified by inthdr, in the interrupt vector table specified by inhno. For the processors in which the interrupt vector table is not implemented, the inthdr is set to the interrupt handler table defined as the variable array. The content of inhno may change with the type of processor (Interrupt vactor numbers are same).

The structure of the interrupt handler information packet Is as shown below. Depending on the type of processor, interrupt mask imask is added at the time of start of interrupt handler.

```
typedef struct t_dinh
{       ATR inhatr;          Interrupt handler attribute
        FP inthdr;           Pointer to the function used as the interrupt handler
        UINT imask;          Interrupt mask (depending on the processor)
}T_DINH;
```

Although value of inhatr is not referred in NORTi, in order to keep the compatibility with other μITRON OS, please specify inhatr as TA_HLNG that shows that task is described in high-level language.

Since it is dependent on the processor, the interrupt handler definition sample is separated from kernel and is described in n4ixxx.c file. User need to customize def_inh so as to match correctly with user’s system. As per μITRON specification, interrupt handler is undefined when pk_dinh is specified as NULL. However, since such functionality is useless in Embedded system, user may not define such functionality for def_inh.

This system call does not function when an interrupt vector table is defined in ROM domain. Please define the interrupt handler address directly to the interrupt vector table.

Return

E_OK	Successful termination
E_PAR	The interrupt definition number dintno is out of the range. *

ent_int

Function Interrupt handler start

Declaration void ent_int(void);

Description This system call saves the registers at the time of interrupt generation, and also changes the stack pointer to the domain reserved for interrupt handler operations. This system call must be called at the start of interrupt handler function.

Since the stack pointer is moved, auto variables cannot be defined at the entry of interrupt handler. User may use static variables, or may call different function from interrupt handler and use auto variables in that function.

Moreover, in some cases just before calling ent_int, there may be an assembly code developed that destroys the register contents before they are saved inside ent_int. In such cases, please control this code deployment by compiler optimization effect etc or by calling a separate function from interrupt handler and by processing the actual handler operation in the that function.

In the interrupt routine which does not include multitasking operation (having priority above the priority of other interrupt handler that is involved in multitasking operation), it is okay even when ent_int and ret_int (described next) system calls are not used. In that case, please use either the compiler extended functions for the interrupt function, or please perform the saving / restoring of registers by uniquely described assembly code.

Return None

Note It is a system call exclusive to NORTi for describing interrupt handler in C.

Example

```
void func(void)    ← (Note)During optimization inline assembler should be off
{
        int c;
                :
}

INTHDR inthdr(void)
{
        ent_int();
        func();
        ret_int();
}
```

ret_int

Function Return from the interrupt handler

Declaration void ret_int(void);

Description This system call terminates interrupt handlers. Be sure to call at the end of interrupt handlers.

The system calls issued inside interrupt handlers to switch tasks is delayed till this ret_int is issued (delayed dispatch).

Return None (not returning to the calling source)

Example

```
INTHDR inthdr(void)
{
        ent_int();
                :
        ret_int();
}
```

chg_ims

Function	Interrupt mask change
Declaration	ER chg_ims(UINT imask); imask Interrupt mask value

Description

This system call changes the interrupt mask of processors to the value specified by imask. In case of the processors that possess only two conditions, interrupt prohibition and interrupt permission, the former is specified by imask!=0 and the latter is specified by imask=0.

In processors that possess level interrupt functions, the system call specifies the interrupt mask level in imask (interrupts permitted with 0 and interrupts prohibited with 1 and more). The chg_ims system call does not check the range of imask value.

In some system calls issued with interrupts prohibited, if switching tasks is necessary, it is done when the interrupt is permitted after chg_ims(0) is issued (this is a delayed dispatch).

Return	E_OK	Successful termination

get_ims

Function	Interrupt mask reference	
Declaration	ER get_ims(UINT *p_imask);	
	p_imask	Pointer to a location where an interrupt mask value is stored

Description

The get_ims system call references the interrupt mask of the processors and returns it to *p_imask.

In processors that possess only two conditions, interrupt prohibition and interrupt permission, the former is indicated by *p_imask=1 and the latter is indicated by *p_imask=0.

In processors that possess level interrupt functions, the interrupt mask level is indicated by the value in *p_imask.

Return	E_OK	Successful termination

vdis_psw

Function Status register's interrupt mask setting

Declaration UINT vdis_psw(void);

Description The vdis_psw system call sets up the interrupt mask of processor's status register in interrupt prohibited conditions. In processors that possess level interrupt functions, this system call sets it up to the highest interrupt level and prohibits all interrupts.

This system call returns the status register values for processors before this operation as return values.

Return Status register value at the processor before interrupt prohibition

Note This is a NORTi unique system call. It is convenient to execute temporary interrupt prohibitions combining with vset_psw. This system call can be issued also from an interrupt routine with higher priority than the kernel.

Example

```
void func(void)
{
    UINT psw;

    psw = vdis_psw();          Interrupt prohibition
        :
    vset_psw(psw);             Interrupt prohibition/permission state is restored
        :
}
```

In order to realize the same thing by chg_ims...

```
void func(void)
{
    UINT imask;

    get_ims(&imask);           Interrupt mask level is read
    chg_ims(7);                Interrupt is inhibited
        :
    chg_ims(imask);            Interrupt is allowed again
}
```

vset_psw

Function Status register setting

Declaration void vset_psw(UINT psw);

psw Processor status register value

Description The vset_psw system call sets up the status registers in processors according to values specified by psw. When the return value of the vdis_psw system call is set up to psw, interrupt masks are completely restored.

This system call is different from chg_ims(0) as this system call does not execute even if there is a delayed dispatch. Therefore no system calls that carry out task switching should be issued between vdis_psw and vset_psw.

Return None

Note This is a system call unique to NORTi. This system call can operate not only interrupt mask bits but also all bits of status registers. It can also be issued from an interrupt routine with priority higher than kernel.

Example

```
void func(void)
{
        UINT psw;

        psw = vdis_psw();
                :
        vset_psw(psw | 0x8000);
                :
}
```

cre_isr

Function Create an Interrupt Service Routine

Declaration ER cre_isr(ID isrid, const T_CISR *pk_cisr);

isrid	Interrupt Service routine ID
pk_cisr	Pointer to the location to store Interrupt Service routine creation information packet

Description The cre_isr system call sets the interrupt service routine specified by isr, to the interrupt number specified by intno. For the processors in which the interrupt vector table is not implemented, the isr is set to the interrupt handler table defined as the variable array. The content of intno may change with the type of processor. Interrupt vector number and interrupt factor number are common.

Following is the structure for interrupt service routine creation packet

```
typedef struct t_cisr
{   ATR istatr;        Interrupt service routine attribute
    VP_INT exinf;      Extended information
    INTNO intno;       Interrupt number
    FP isr;            Interrupt Service Routine address
    UINT imask;        Interrupt mask (Processor related)
}T_CISR;
```

Although value of istatr is not referred in NORTi, in order to keep the compatibility with other μITRON OS, please specify inhatr as TA_HLNG that shows that task is described in high-level language.

Since it is dependent on the processor, the interrupt handler definition sample is separated from kernel and is described in n4ixxx.c file. User need to customize def_inh so as to match correctly with user’s system.

In case of interrupt service routine, it is not necessary for OS to call ent_int / ret_int in order to perform enterance / exit processing from the interrupt handler. Since there is no restriction in interrupt handler such as prohibition of auto variables etc, it can be described as a general C function. However, it is not possible to use the interrupt service routine to handle interrupt of priority higher than Kernel level.

In the attached samples, the interrupt handler number specified by def_inh is same as the interrupt handler number specified ny cre_isr. Multiple interrupt service routines can be attached to the same interrupt number.

Return	E_OK	Successful termination
	E_PAR	Interrupt number intno is outside range *
	E_ID	ID is outside range *
	E_SYS	The memory for a management block is not securable. **

acre_isr

Function Create an Interrupt Service Routine (automatic ID allocation)

Declaration ER_ID acre_isr(const T_CISR *pk_cisr);

pk_cisr Pointer to the location to store Interrupt Service routine creation information packet

Description This system call allocates the highest ID value searched from non-generated Interrupt Service routine ID values. System call will return with E_NOID error when the ID allocation fails. Except above the other part is same as cre_isr system call.

Return The interrupt service routine ID is assigned when it is positive value.

E_NOID	Insufficient value for interrupt service routine ID
E_CTX	It is issued from an interrupt handler *
E_SYS	The memory for a management block is not securable. **

Example

```
ID ID_isr1;
extern void sioist(VP_INT);
const T_CISR cisr1 = {TA_HLNG, NULL, INT_SIO1, sioisr, 0X07};

TASK task1(void)
{
    ER_ID ercd;
        :
    ercd = acre_isr(&cisr1);
    if(ercd > 0)
    ID_isr1 = ercd;
}
```

del_isr

Function	Deletion of interrupt service routine

Declaration ER del_isr(ID isrid);

isrid	Interrupt service routine ID

Description The interruption service routine specified by isrid is deleted.

Return

E_OK	Successful termination
E_ID	ID is outside valid range*
E_NOEXS	Object does not exist
E_CTX	It is issued from an interrupt handler *

ref_isr

Function Refer to the state of the interrupt service routine

Declaration ER ref_isr(ID isrid, T_RISR *pk_risr);

isrid	Interrupt service routine ID
pk_risr	The pointer to the location which stores the interrupt service routine state information packet

Description This system call returns the state of the interrupt service routine specified by isrid, to *pk_risr.

The structure of the interrupt service routine state packet is as shown below.

```
typedef struct t_risr
{       INTNO intno;        Interrupt number
        UINT imask;         Interrupt mask (processor related)
}T_RISR;
```

Return

E_OK	Successful termination
E_ID	ID is outside valid range*
E_NOEXS	Object does not exist

5.12 Memory pool management functions (Variable length)

cre_mpl

Function Create variable length memory pool

Declaration ER cre_mpl(ID mplid, const T_CMPL *pk_cmpl);

mplid	Variable length memory pool ID
pk_cmpl	The pointer to the variable length memory pool creation information packet

Description The cre_mpl system call creates the variable-length memory pool specified by mplid. A variable-length memory pool management block is dynamically allocated from the system memory. When pk_cmpl ->mpl is NULL, only the size specified by pk_cmpl->mplsz bytes is dynamically allocated from the memory reserved for the memory pool.

When a variable-length memory pool creation information packet is placed in memory domain other than ROM (i.e. when a const data type is not attached), the creation information packet data is copied to the system memory.

Following is the structure of the variable length memory pool creation information packet.

```
typedef struct t_cmpl
{       ATR mplatr;          Variable length memory pool attribute
        SIZE mplsz;          Size of whole memory pool (byte count)
        VP mpl;              Memory pool head address or NULL
        B *name;             The pointer to the variable pool name string (optional)
}T_CMPL;
```

Please put the following value into mplatr, the variable length memory pool attribute.

TA_TFIFO	Acquisition waiting task processing in the order of arrival (FIFO)
TA_TPRI	Acquisition waiting task processing in the order of priority.

When the memory pool domain is allocated by the user program, please set the block start address and byte size in pk_cmpl-> mpl and pk_cmpl->mplsz respectively. Since there is an overhead by OS, all of the memory size cannot be allocated to user program.

The macro function TSZ_MPL(bcnt, blksz) returns the total size required for allocation of bcnt number of data blocks each of size blksz.

Since name is for debugger correspondence, please set “” or NULL when none is selected. You may omit name when creation information structure object is defined with initial value.

Return	E_OK	Successful termination
	E_ID	Variable length memory pool ID is outside valid range*
	E_OBJ	Variable length memory pool is already created
	E_CTX	The command issued from an interrupt handler*
	E_SYS	Insufficient system memory for management block**
	E_NOMEM	Insufficient memory for memory pool**

Note1 Every single memory block acquisition, “sizeof(int *)” bytes only is used for OS management purpose, i.e. 4 bytes for CPU which has data domain address space of 32bit and 2 bytes for CPU which has data domain address space of 16bit. Therefore, please consider above mentioned part for OS management for calculation of mplsz. In addition, in order to maintain alignment with “sizeof(int *)” bytes, the domain could be excessive to the size.

Note2 When memory pool acquision and release is called repeatedly, the memory pool memory gets fragmenized i.e. the size of continuous free memory gets reduced. (There is no function to defragment the memory pool.)

Example

```
#define ID_mpl1    1
const T_CMPL cmpl1 = {TA_TFIFO, 1024, NULL};

TASK task1(void)
{
        ER ercd;
                :
        ercd = cre_mpl(ID_mpl1, &cmpl1);
                :
}
```

acre_mpl

Function Create variable length memory pool (Automatic ID allocation)

Declaration ER_ID acre_mpl(const T_CMPL *pk_cmpl);

pk_cmpl	The pointer to the variable length memory pool creation information packet

Description This system call allocates the highest ID value searched from non-generated variable-length memory pool ID values. System call will return with E_NOID error when the ID allocation fails. Except above the other part is same as cre_mpl system call.

Return A positive value indicates the allocated ID for variable length memory pool.

E_NOID	Insufficient ID for variable length memory pool
E_CTX	The command issued from an interrupt handler*
E_SYS	Insufficient system memory for management block**
E_NOMEM	Insufficient memory for memory pool**

Example

```
ID ID_mpl1;
const T_CMPL cmpl1 = {TA_TFIFO, 1024, NULL };

TASK task1(void)
{
    ER_ID ercd;
        :
    ercd = acre_mpl(&cmpl1);
    if(ercd > 0)
    ID_mpl1 = ercd;
        :
}
```

del_mpl

Function	Delete variable length memory pool
Declaration	ER del_mpl(ID mplid); mplid Variable length memory pool ID

Description

The del_mpl system call deletes a variable-length memory pool specified by mplid. The variable-length memory pool management block is released to the system memory. In case if the OS did the allocation of memory pool domain, the memory pool domain is released back to the memory-pool memory.

When a task is waiting this variable length memory pool for memory allocation, the system call releases this task from waiting. The task, whose wait was released, returns an E_DLT error indicating that the variable length memory pool was deletion while the task was waiting for it.

Return

E_OK	Successful termination
E_ID	Variable length memory pool ID is outside valid range*
E_NOEXS	Variable length memory pool is not created
E_CTX	The command issued from an interrupt handler*

Example

```
#define ID_mpl1    1

TASK task1(void)
{
            :
    del_mpl(ID_mpl1);
            :
}
```

get_mpl

Function	Acquisition of variable-length memory pool

Declaration ER get_mpl(ID mplid, UINT blksz, VP *p_blk);

mplid	Variable length memory pool ID
blksz	Memory block size (Byte count)
p_blk	A pointer to a location which stores memory block pointer.

Description The get_mpl system call acquires memory block of size blksz from the variable-length memory pool specified by mplid and returns the pointer of that memory block to *p_blk. Zero clearing of acquired memory block is not performed. The block data is undefined.

When the emprty block size in variable size memory pool is insufficient, then the task which had issued this system call will be connected to the queue waiting for the variable size memory pool.

The minimum value for the memory block size blksz is 1 byte. However for processor which requires word (4 bytes) alignment, blfsz should be integer multiple of size of int (in case of non-integer or fractional multiple ratio, it is realigned inside OS).

In order to acquire a memory block of size blksz, the variable length memory pool should have continuous empty free space of “blksz + sizeof(int)” bytes.

The system does not processing priority for smaller size of the requested memory block.

Return

E_OK	Successful termination
E_ID	Variable length memory pool ID is outside valid range*
E_NOEXS	Variable length memory pool is not created
E_CTX	Issued from the non-task context, or waiting in dispatch prohibited state*
E_RLWAI	Waiting state was released forcibly (rel_wai was issued while waiting)
E_DLT	Variable length memory pool was deleted while waiting for it

Note1 p_blk is a pointer to pointer i.e. double pointer.

Note2 It is same as tget_mpl(mplid, blksz, p_blk, TMO_FEVR).

Example

```
#define ID_mpl1    1

TASK task1(void)
{
        B *blk;
                :
        get_mpl(ID_mpl1, 256, (VP *)&blk);
        blk[0] = 0;
        blk[1] = 1;
:
}
```

pget_mpl

Function Acquisition of variable-length memory pool (Polling mode)

Declaration ER pget_mpl(ID mplid, UINT blksz, VP *p_blk);

mplid	Variable length memory pool ID
blksz	Memory block size (Byte count)
p_blk	A pointer to a location which stores memory block pointer.

Description Following are the differences from get_mpl.

When there is insufficient memory block in variable size memory pool, then instead of waiting in queue, this system call returns back with E_TMOUT error.

Return

E_OK	Successful termination
E_ID	Variable length memory pool ID is outside valid range*
E_NOEXS	Variable length memory pool is not created
E_TMOUT	Polling failure
E_CTX	The command issued from an interrupt handler*

Note1 p_blk is a pointer to pointer i.e. double pointer.

Note2 It is same as tget_mpl(mplid, blksz, p_blk, TMO_POL).

Example

```
#define ID_mpl1    1

TASK task1(void)
{
        B *blk;
        ER ercd;
                :
        ercd = pget_mpl(ID_mpl1, 256, (VP *)&blk);
        if (ercd == E_OK)
                :
}
```

tget_mpl

Function Acquisition of variable-length memory pool (Timeout available)

Declaration ER tget_mpl(ID mplid, UINT blksz, VP *p_blk, TMO tmout);

mplid	Variable length memory pool ID
blksz	Memory block size (Byte count)
p_blk	A pointer to a location which stores memory block pointer
tmout	Timeout value

Description Following are the differences from get_mpl.

When a memory block of required size is not acquired even after the time specified by tmout has passed, a time-out error E_TMOUT is returned back.

When this system call is issued with tmout=TMO_POL (=0), the call executes similar to pget_mpl, i.e. it does not perform waiting. For tmout=TMO_FEVR (=-1), this system call runs same as get_mpl, i.e. there is no timeout.

Return

E_OK	Successful termination
E_ID	Variable length memory pool ID is outside valid range*
E_NOEXS	Variable length memory pool is not created
E_CTX	Issued from the non-task context, or waiting in dispatch prohibited state*
E_RLWAI	Waiting state was released forcibly (rel_wai was issued while waiting)
E_DLT	Variable length memory pool was deleted while waiting for it
E_TMOUT	Timeout

Note p_blk is a pointer to pointer i.e. double pointer.

Example

```
#define ID_mpl1    1

TASK task1(void)
{
        B *blk;
        ER ercd;
                :
        ercd = tget_mpl(ID_mpl1, 256, (VP *)&blk, 100/MSEC);
        if (ercd == E_OK)
                :
}
```

rel_mpl

Function Release variable-length memory block.

Declaration ER rel_mpl(ID mplid, VP blk);

mplid	Variable length memory pool ID
blk	Memory block pointer

Description Memory block pointed by blk is returned to the variable-length memory pool specified by mplid.

If there is a task, which is waiting for memory-block acquisition from this variable-length memory pool, when the empty size of the memory pool as a result of the memory block release, is higher than the size requested by heading task in waiting queue, then the memory block is allocated to that task and is released from wait.

In some cases it is possible that by single call to this function, two or more tasks from queue waiting for memory block acquisition are released. In such case, the memory blocks are allocated sequentially starting from the top of the queue. The task issuing this system call will not change to waiting state.

Always make sure that the memory pool is released back to the same source from where it was acquired. Memory leak phenomenon may occur when the memory pool is not released before termination of used objects such as task etc.

Return

E_OK	Successful termination
E_PAR	Returned to different memory pool
E_ID	Variable length memory pool ID is outside valid range*
E_NOEXS	Variable length memory pool is not created
E_CTX	The command issued from an interrupt handler*

Example

```
#define ID_mpl1    1

TASK task1(void)
{
        B *blk;
                :
        get_mpl(ID_mpl1, 256, (VP *)&blk);
                :
        rel_mpl(ID_mpl1, (VP)blk);
                :
}
```

ref_mpl

Function Get reference of variable-length memory pool state.

Declaration ER ref_mpl(ID mplid, T_RMPL *pk_rmpl);

mplid	Variable length memory pool ID
pk_rmpl	A pointer to the location which stores variable-length memory pool state

Description A state of a variable-length memory pool specified by mplid is returned to *pk_rmpl.

A structure of a variable-length memory pool state packet is as follows.

```
typedef struct t_rmpl
{       ID wtskid;          ID of the waiting task or TSK_NONE
        SIZE fmplsz;        Total free memory size (Byte count)
        UINT fblksz;        Maximum memory block size available (Byte count)
}T_RMPL;
```

When a waiting task exists, ID of the first waiting task is returned. When there is no waiting task, TSK_NONE is returned.

Return

E_OK	Successful termination
E_ID	Variable length memory pool ID is outside valid range
E_NOEXS	Variable length memory pool is not created

Example

```
#define ID_mpl1    1

TASK task1(void)
{
        T_RMPL rmpl;
                :
        ref_mpl(ID_mpl1, &rmpl);
        if (rmpl.fmplsz >= 256 + sizeof(int))
                :
}
```

5.13 Memory pool management functions (Fixed length)

cre_mpf

Function Create fixed-length memory pool

Declaration ER cre_mpf(ID mpfid, const T_CMPF *pk_cmpf);

mpfid Fixed-length memory pool ID

pk_cmpf A pointer to a fixed-length memory pool creation information packet

Description The cre_mpf system call creates the fixed-length memory pool specified by mpfid. A fixed-length memory pool management block is dynamically allocated from the system memory. When pk_cmpf ->mpf is NULL, only the size specified by blkcnt x blfsz bytes is dynamically allocated from the memory reserved for the memory pool. When the memory pool domain is allocated by the user program, please set the block start address in pk_cmpf-> mpf.

Following is the structure of the fixed length memory pool creation information packet.

```
typedef struct t_cmpf
{       ATR mpfatr;             Fixed-length memory pool attribute
        UINT blkcnt;            Total number of blocks in the memory pool
        UINT blfsz;             Fixed-length memory block size (Byte count)
        VP mpf;                 Memory pool start address, or NULL
        B *name;                Pointer to the memory pool name (optional)
}T_CMPF;
```

Following are the valid set values for mpfatr, i.e. fixed-length memory pool attribute.

TA_TFIFO Processing of the acquision waiting task is in the order of arrival (FIFO)

TA_TPRI Processing of the acquision waiting task is in the order of task priority

The minimum value of the memory block size, i.e. blksz, is more than the pointer size of the processing system. Moreover, for processors that need word alignment, blfsz should be integer multiple of size of int (in case of non-integer or fractional multiple ratio, it is realigned inside OS).

The size of the memory pool, consumed by acquision of memory block of size blksz, is equal to blksz. Hence there is no memory waste.

Since name is for debugger correspondence, please set “” or NULL when none is selected. You may omit name when creation information structure object is defined with initial value.

Return	E_OK	Successful termination
	E_ID	Fixed-length memory pool ID is outside valid range*
	E_OBJ	The fixed length memory pool is already created
	E_CTX	Command issued from an Interrupt handler*
	E_SYS	Insufficient system memory for management block**
	E_NOMEM	Insufficient memory for memory pool**

Example

```
#define ID_mpf1    1
const T_CMPF cmpf1 = {TA_TFIFO, 10, 256, NULL};

TASK task1(void)
{
        ER ercd;
                :
        ercd = cre_mpf(ID_mpf1, &cmpf1);
                :
}
```

acre_mpf

Function Create fixed-length memory pool (Automatic ID allocation)

Declaration ER_ID acre_mpf(const T_CMPF *pk_cmpf);

pk_cmpf	A pointer to a fixed-length memory pool creation information packet

Description This system call allocates the highest ID value searched from non-generated fixed-length memory pool ID values. System call will return with E_NOID error when the ID allocation fails. Except above the other part is same as cre_mpf system call.

Return A positive value indicates the allocated ID for fixed length memory pool.

E_NOID	Insufficient ID for fixed length memory pool
E_CTX	Command issued from an Interrupt handler *
E_SYS	Insufficient system memory for management block**
E_NOMEM	Insufficient memory for memory pool**

Example

```
ID ID_mpf1;
const T_CMPF cmpf1 ={TA_TFIFO, 10, 256, NULL};

TASK task1(void)
{
    ER_ID ercd;
        :
    ercd = acre_mpf(&cmpf1);
    if(ercd > 0)
        ID_mpf1 = ercd;
        :
}
```

del_mpf

Function Remove/Delete fixed-length memory pool

Declaration ER del_mpf(ID mpfid);

mpfid Fixed-length memory pool ID

Description The del_mpl system call deletes a fixed-length memory pool specified by mpfid. The fixed-length memory pool management block is released to the system memory. In case if the OS did the allocation of memory pool domain, the memory pool domain is released back to the memory-pool memory.

When a task is waiting this fixed length memory pool for memory allocation, the system call releases this task from waiting. The task, whose wait was released, returns an E_DLT error indicating that the fixed-length memory pool was deletion while the task was waiting for it.

Return

E_OK	Successful termination
E_ID	A fixed-length memory pool ID is outside valid range*
E_NOEXS	A fixed-length memory pool is not yet created.
E_CTX	Command issued from an Interrupt handler *

Example

```
#define ID_mpf1    1

TASK task1(void)
{
            :
      del_mpf(ID_mpf1);
            :
}
```

get_mpf

Function	Acquisition of fixed-length memory pool

Declaration ER get_mpf(ID mpfid, VP *p_blf);

mpfid	Fixed-length memory pool ID
p_blf	A pointer to a location which stores memory block pointer.

Description The get_mpf system call acquires single memory block from the fixed-length memory pool specified by mpfid and returns the pointer of that memory block to *p_blf. The size of the memory block is fixed to blfsz, which was set at the time of fixed-length memory pool creation. Zero clearing of acquired memory block is not performed. The block data is undefined.

When there is no vacant block in fixed size memory pool, then the task which had issued this system call will be connected to the queue waiting for the fixed size memory pool.

Return

E_OK	Successful termination
E_ID	A fixed-length memory pool ID is outside valid range*
E_NOEXS	A fixed-length memory pool is not yet created.
E_CTX	Issued from the non-task context, or waiting in dispatch prohibited state*
E_RLWAI	Waiting state was released forcibly (rel_wai was issued while waiting)
E_DLT	Fixed length memory pool was deleted while waiting for it

Note1 p_blf is a pointer to pointer i.e. double pointer.

Note2 It is same as tget_mpf(mpfid, p_blf, TMO_FEVR).

Example

```
#define ID_mpf1    1

TASK task1(void)
{
        B *blf;
                :
        get_mpf(ID_mpf1, (VP *)&blf);
        blf[0] = 0;
        blf[1] = 1;
                :
}
```

pget_mpf

Function Acquisition of fixed-length memory pool (Polling mode)

Declaration ER pget_mpf(ID mpfid, VP *p_blf);

mpfid	Fixed-length memory pool ID
p_blf	A pointer to a location which stores memory block pointer.

Description Following are the differences from get_mpf.

When there is no vacant block in fixed size memory pool, then instead of waiting in queue, this system call returns back with E_TMOUT error.

Return

E_OK	Successful termination
E_ID	A fixed-length memory pool ID is outside valid range*
E_NOEXS	A fixed-length memory pool is not yet created.
E_TMOUT	Polling failure

Note1 p_blf is a pointer to pointer i.e. double pointer.

Note2 It is same as tget_mpf(mpfid, p_blf, TMO_POL).

Example

```
#define ID_mpf1    1

TASK task1(void)
{
        B *blf;
        ER ercd;
                :
        ercd = pget_mpf(ID_mpf1, (VP *)&blf);
        if(ercd == E_OK)
                :
}
```

tget_mpf

Function Acquisition of fixed-length memory pool (Timeout available)

Declaration ER tget_mpf(ID mpfid, VP *p_blf, TMO tmout);

mpfid	Fixed-length memory pool ID
p_blf	A pointer to a location which stores memory block pointer.
tmout	Timeout Value

Description Following are the differences from get_mpf.

When a memory block cannot be gained even after the time specified by tmout has passed, a time-out error E_TMOUT is returned back.

When this system call is issued with tmout=TMO_POL (=0), the call executes similar to pget_mpf, i.e. it does not perform waiting. For tmout=TMO_FEVR (=-1), this system call runs same as get_mpf, i.e. there is no timeout.

Return

E_OK	Successful termination
E_ID	A fixed-length memory pool ID is outside valid range*
E_NOEXS	A fixed-length memory pool is not yet created.
E_CTX	Issued from the non-task context, or waiting in dispatch prohibited state*
E_RLWAI	Waiting state was released forcibly (rel_wai was issued while waiting)
E_DLT	Fixed length memory pool was deleted while waiting for it
E_TMOUT	Timeout error

Example

```
#define ID_mpf1    1

TASK task1(void)
{
        B *blf;
        ER ercd;
                :
        ercd = tget_mpf(ID_mpf1, (VP *)&blf, 100/MSEC);
        if(ercd == E_OK)
                :
}
```

rel_mpf

Function Release Fixed-length memory block.

Declaration ER rel_mpf(ID mpfid, VP blf);

mpfid	Fixed-length memory pool ID
blf	Memory block pointer

Description Memory block pointed by blf is returned to the fixed-length memory pool specified by mpfid. If there is a task, which is waiting for memory-block acquisition from this fixed-length memory pool, a memory block will be allocated to the waiting task (top in waiting queue), and waiting will be canceled.

Unlike variable-length memory block, the memory-block acquisition waiting of two or more tasks by single return is not canceled.

The task, which published this system call, will not change to a waiting state. Please be sure to return memory block to the original memory pool.

Return

E_OK	Successful termination
E_PAR	Release of different memory pool.
E_ID	A fixed-length memory pool ID is outside valid range*
E_NOEXS	A fixed-length memory pool is not yet created.

Example

```
#define ID_mpf1    1

TASK task1(void)
{
        B *blf;
                :
        get_mpf(ID_mpf1, (VP *)&blf);
                :
        rel_mpf(ID_mpf, (VP)blf);
                :
}
```

ref_mpf

Function Get reference of fixed-length memory pool state.

Declaration ER ref_mpf(ID mpfid, T_RMPF *pk_rmpf);

mpfid	Fixed-length memory pool ID
pk_rmpf	A pointer to the location which stores fixed-length memory pool state

Description A state of a fixed-length memory pool specified by mpfid is returned to *pk_rmpf.

A structure of a fixed-length memory pool state packet is as follows.

```
typedef struct t_rmpf
{       ID wtskid;          ID of the waiting task or TSK_NONE.
        UINT fblkcnt;       The number of empty memory blocks.
}T_RMPF;
```

When a waiting task exists, ID of the first waiting task is returned. When there is no waiting task, TSK_NONE is returned.

Return

E_OK	Successful termination
E_ID	A fixed-length memory pool ID is outside of valid range.
E_NOEXS	A fixed-length memory pool is not yet created.

Example

```
#define ID_mpf1    1

TASK task1(void)
{
        T_RMPF rmpf;
                :
        ref_mpf(ID_mpf1, &rmpf);
        if(rmpf.fblkcnt > 0)
                :
}
```

5.14 Time management functions

set_tim

Function System time setup

Declaration ER set_tim(SYSTIM *p_systim);

p_systim The pointer to the present time packet

Description The set_tim system call changes the system clock executing time management to the value specified by *p_systim.

The structure of a time packet is as follows.

```
typedef struct
{       H utime;            Higher 16 bits
        UW ltime;           Lower 32 bits
}SYSTIM;
```

The system time set by set_tim is the count which increments every periodic interrupt. Therefore the system clock is the data which is counting the number of periodic interrupts. It is necessary to perform time conversion to a unit such as msec in user program.

As opposed to expressing the system clock as the absolute time which is cleared to 0 at the time of system starting and then counting up, the system time is a relative time initialized by set_tim. Since the time event handler takes the system clock as the standard clock, it is not affected by set_tim.

Return E_OK Successful termination

Example

```
SYSTIM tim;
        :
tim.utime = 0;
tim.ltime = 12345L;
set_tim(&tim);
        :
```

get_tim

Function Refer to system time.

Declaration ER get_tim(SYSTIM *p_systim);

pk_systim The pointer to the location which stores the present time packet

Description The present value of system time is returned to *pk_systim.

The structure of time packet is same as that of the set_tim system call.

```
typedef struct
{       H utime;        Higher 16 bits
        UW ltime;       Lower 32 bits
}SYSTIM;
```

System time is the data is the count of cyclic interrupt. By the user side, it is necessary to perform conversion with the unit of time, such as msec.

Return E_OK Successful termination

Example

```
SYSTIM tim;
        :
get_tim(&tim);
if(tim.ltime == 10000L)
        :
```

cre_cyc

Function Creation of the cyclic handler

Declaration ER cre_cyc(ID cycid, const T_CCYC *pk_ccyc);

cycid Cyclic handler ID

pk_ccyc The pointer to a cyclic handler creation information packet

Description The cre_cyc system call creates the periodic cyclic handler specified by cycid. A peridoc cyclic handler management block is dynamically allocated from the system memory.

Following is the structure of cyclic handler creation information packet.

```
typedef struct t_ccyc
{   ATR cycatr;         Cyclic handler attribute
    VP_INT exinf;       Extended information
    FP cychdr;          Pointer to the function used as a cyclic handler
    RELTIM cyctim;      Cyclic handler activation time
    RELTIM cycphs;      Cyclic handler activation phase
}T_CCYC;
```

Following are the valid inputs for cycatr. Please specify only TA_HLNG attribute, when TA_STA and TA_PHS are unnecessary.

TA_HLNG In order to maintain the compatibility with other μITRON based OS, please set TA_HLNG, which shows that the handler is described with the high-level language.

TA_STA Handler is in operational state when it is created

TA_PHS Activation phase of the handler is preserved

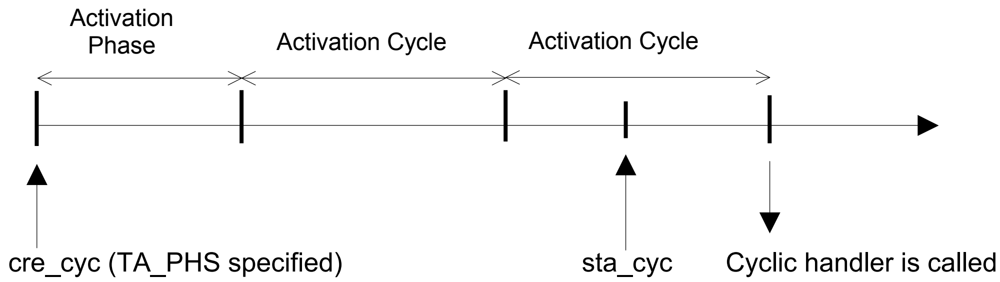
When the activation phase is not preserved, a cycle is initialized when the handler operation is started. Therefore, the first cycle of handler always starts from the start of handler operation. When the activation phase is preserved, after the creation of handler the clocking is continued regardless of operational state of cyclic handler.

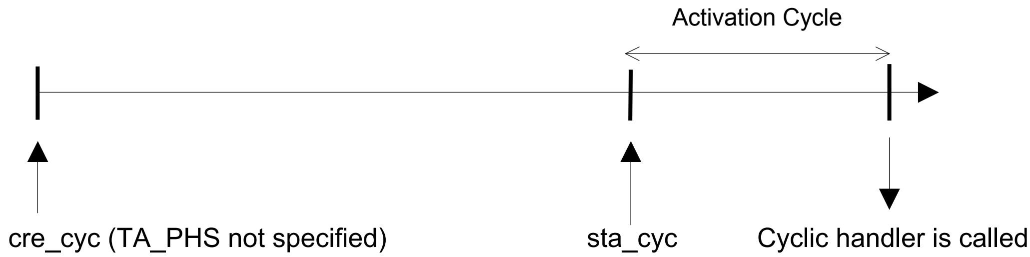
The value specified to exinf is passed as the first parameter at the time of handler starting.

cychdr is the pointer to the function which is used as the periodic handler. Please describe the periodic handler as the void type function.

cyctim is the interval time of the activation cycle. The system clock interrupt cycle is the time unit for handler operation.

Please set cycphs as the time from start of handler operation and until it is activated for first time. From the second cycle onwards, cyctim is the interval time.





Return	E_OK	Successful termination
	E_ID	The cyclic handler ID is outside valid range*
	E_PAR	The cyclic handler active state is illegal*
	E_CTX	The command issued from an interrupt handler*
	E_SYS	Insufficient system memory for management block**

* For NORTi Kernel previous to 4.05.00, E_ID was incorrectly defined as E_PAR.

Example

```
#define ID_cyc1    1
extern void cyc1(VP_INT);
const T_CCYC ccyc1 ={TA_HLNG|TA_STA, NULL, cyc1, 10, 5};

TASK task1(void)
{
       ER ercd;
              :
       ercd = cre_cyc(ID_cyc1, &ccyc1);
              :
}
```

acre_cyc

Function Creation of the cyclic handler (automatic ID allocation)

Declaration ER_ID acre_cyc(const T_CCYC *pk_ccyc);

Pk_ccyc	The pointer to a cyclic handler creation information packet

Description The highest value from the non-generated cyclic handler ID is searched and assigned. An E_NOID error is returned when the cyclic handler ID is not assigned. Other than this rest is the same as cre_cyc.

Return The cyclic handler ID assigned if a positive value

E_NOID	Cyclic handler ID is incorrect
E_PAR	A cyclic handler activity state is incorrect. *
E_CTX	Issued from an interrupt handler *
E_SYS	Memory for a management block is not securable. **

Example

```
ID ID_cyc1;
extern void cyc1(VP_INT);
const T_CCYC ccyc1 = {TA_HLNG|TA_STA, NULL, cyc1, 10, 5};

TASK task1(void)
{
    ER_ID ercd;
        :
    ercd = acre_cyc(&ccyc1);
    if(ercd > 0)
        ID_cyc1 = ercd;
        :
}
```

del_cyc

Function	Deletion of the cyclic handler
Declaration	ER del_cyc(ID cycid); cycid Cyclic handler ID
Description	The cyclic handler specified by cycid is deleted. A cyclic handler management block is released to system memory.

Return

E_OK	Successful termination
E_ID	The cyclic handler ID is outside range. *
E_NOEXS	The cyclic handler does not exist.
E_CTX	Issued from an interrupt handler *

Example

```
ID ID_cyc1;

TASK task1(void)
{
    ER ercd;
        :
    ercd = del_cyc(ID_cyc1);
        :
}
```

sta_cyc

Function	Start Cyclic handler operation

Declaration ER sta_cyc(ID cycid);

cycid Cyclic handler ID

Description The sta_cyc system call brings the cyclic handler specified by cycid to the operation state. When there is no TA_PHS specification, handler starts after starting cycle passes from a sta_cyc call. When TA_PHS is specified, nothing is done if it is already in operating state. When TA_PHS is specified and it is in stopped state, it is brought to the activation state without changing the clock. When TA_PHS is specified, renewal of startup time is performed irrespective of the ability to start.

Return		
	E_OK	Successful termination
	E_ID	The cyclic handler ID is outside valid range*
	E_NOEXS	The cyclic handler does not exist.

* For NORTi Kernel previous to 4.05.00, E_ID was incorrectly defined as E_PAR.

stp_cyc

Function Stops Cyclic handler operation

Declaration ER stp_cyc(ID cycid);

cycid Cyclic handler ID

Description The stp_cyc system call brings the cyclic handler specified by cycid to the non-operational state. If a handler that is already stopped is specified, then nothing is done.

When TA_PHS is specified during creation the renewal of the activation clock is continued.

Return		
	E_OK	Successful termination
	E_ID	The cyclic handler ID is outside range. *
	E_NOEXS	The cyclic handler does not exist.

ref_cyc

Function: Refers to Cyclic handler

Declaration: ER ref_cyc(ID cycid, T_RCYC *pk_rcyc);

cycid	Cyclic handler ID
pk_rcyc	Pointer to the location which stores the cyclic handler condition packet.

Description: The state of the periodic handler specified by cycid is returned to *pk_rcyc.

The structure of a periodic handler state packet is as follows.

```
typedef struct t_rcyc
{       STAT cycstat;         Operating state of a handler
        RELTIM lefttim;       Time left till next activation
}T_RCYC;
```

The following value goes into cycstat according to operating state.

TCYC_STP	The handler is not operating.
TCYC_STA	The handler is operating.

The unit of lefttim is the interrupt cycle of a system clock.

Return

E_OK	Successful termination
E_ID	The cyclic handler ID is outside range.
E_NOEXS	The cyclic handler does not exist.

Example

```
#define ID_cyc    1

TASK task1(void)
{
        T_RCYC rcyc;
                :
        ref_cyc(ID_cyc, &rcyc);
        if(rcyc.cycstat == TCYC_STA)
                :
}
```

cre_alm

Function Alarm handler generation

Declaration ER cre_alm(ID almid, const T_CALM *pk_calm);

almid	Alarm handler ID
pk_calm	The pointer to an alarm handler creation information packet

Description The alarm handler specified by almid is generated. An alarm handler management block is dynamically assigned from system memory.

The structure of an alarm handler generation information packet is as follows.

```
typedef struct t_calm
{   ATR almatr;       Alarm handler attribute
    VP_INT exinf;     Extended information
    FP almhdr;        The pointer to the function used as an alarm handler
}T_CALM;
```

almhdr is a pointer to the function used as an alarm handler. Please describe an alarm handler as a void type function.

Although NORTi does not refer the value of almatr, in order to maintain the compatibility with other μITRON based OS, Please set almatr to TA_HLNG, which shows that the handler is described with the high-level language. The value of exinf is passed as the second argument of an alarm handler.

Return

E_OK	Successful termination
E_ID	An alarm handler ID number is outside range. *
E_PAR	Parameter error *
E_OBJ	An alarm handler is registered. *
E_CTX	Issued from an interrupt handler *
E_SYS	Memory for a management block is not securable. **

Example

```
#define ID_alm1    1
extern void alm1(VP_INT);
const T_CALM calm1 = {TA_HLNG, NULL, alm1};

TASK task1(void)
{
    ER ercd;
        :
    ercd = cre_alm(ID_alm1, &calm1);
        :
}
```

acre_alm

Function Alarm handler generation (automatic ID assignment)

Declaration ER_ID acre_alm(const T_CALM *pk_calm);

pk_calm The pointer to an alarm handler creation information packet

Description The highest value of non-generated alarm handler ID is searched and assigned. An E_NOID error is returned when the alarm handler ID is not assigned. Other than this rest is same as cre_alm.

Return The alarm handler ID assigned when it was a positive value

E_NOID	The alarm handler ID is insufficient.
E_OBJ	An alarm handler is registered. *
E_CTX	Issued from an interrupt handler *
E_SYS	Memory for a management block is not securable. **

Example

```
ID ID_alm1;
extern void alm1(VP_INT);
const T_CALM calm1 = {TA_HLNG, NULL, alm1};

TASK task1(void)
{
    ER_ID ercd;
        :
    ercd = acre_alm(&calm1);
    if(ercd > 0)
    ID_alm1 = ercd;
        :
}
```

del_alm

Function Deletion of an Aralm handler

Declaration ER del_alm(ID almid);

almid Alarm handler ID

Description The alarm handler specified by almid is deleted. An alarm handler management block is released to system memory.

Return

E_OK	Successful termination
E_ID	The alarm handler ID is outside range. *
E_NOEXS	The alarm handler is not generated.
E_CTX	Issued from an interrupt handler *

Example

```
ID ID_alm1;

TASK task1(void)
{
        ER ercd;
                :
        ercd = del_alm(ID_alm1);
                :
}
```

sta_alm

Function	Alarm handler operation start

Declaration ER sta_alm(ID almid, RELTIM almtim);

almid	Alarm handler ID number
almtim	Alarm handler starting time (relative time)

Description The starting time of an alarm handler specified by almid is set as almtim, and operation is started. Starting time is changed into a new value when the handler under operation is specified.

The activation time is the time relative to time when sta_tim was called, taking the timer interrupt interval as a time unit.

Return

E_OK	Successful termination
E_ID	The alarm handler ID is outside range. *
E_NOEXS	The alarm handler is not defined

stp_alm

Function Alarm handler operation stop

Declaration ER stp_alm(ID almid);

almid	Alarm handler ID number

Description The starting time of an alarm handler specified by almid is canceled and changed into the state where it is not operating. Nothing is done when the handler that has already stopped is specified.

Return

E_OK	Successful termination
E_ID	The alarm handler ID is outside range. *
E_NOEXS	The alarm handler is not defined

** For NORTi Kernel previous to 4.05.00, E_ID was incorrectly defined as E_PAR.

ref_alm

Function Refer to alarm handler state.

Declaration ER ref_alm(ID almid, T_RALM *pk_ralm);

almid	Alarm handler ID
pk_ralm	The pointer to the location which stores an alarm handler state packet

Description The state of the alarm handler specified by almid is returned to *pk_ralm.

The structure of an alarm handler state packet is as follows.

```
typedef struct t_ralm
{     STAT almstat;        The state of a handler
      RELTIM lefttim;      Remaining time to start
}T_RALM;
```

The following value returns to almstat.

TALM_STP	The alarm handler is not operating.
TALM_STA	The alarm handler is operating.

The remaining time to start will be returned to lefttim.

Return

E_OK	Successful termination
E_ID	The alarm handler ID is outside range. *
E_NOEXS	The alarm handler is not defined

** For NORTi Kernel previous to 4.05.00, E_ID was incorrectly defined as E_PAR.

Example

```
#define ID_alm1    1

TASK task1(void)
{
      T_RALM ralm;
            :
      ref_alm(ID_alm1, &ralm);
      if(ralm.lefttim > 100/MSEC)
            :
}
```

isig_tim

Function Tick time end notice

Declaration void isig_tim(void);

Description This function informs OS about entry of periodic timer interrupt.
It is exclusively for interrupt handler.

Return none

Note This system call is exclusive to NORTi.

Example

```
INTHDR inthdr(void)
{
ent_int();
isig_tim();
ret_int();
}
```

def_ovr

Function Define overrun handler

Declaration ER def_ovr(const T_DOVR *pk_dovr);

pk_dovr Pointer to the overrun handler definition information packet

Description Overrun handler is defined based on the specified definition information.

Following is the structure of an overrun handler information packet.

```
typedef struct t_dovr
{       ATR ovratr;         Overrun handler attribute
        FP ovrhdr;          Overrun handler address
        INTNO intno;        Cyclic interrupt number to be used
        FP ovrclr;          Star address of the function which clears the interrupt
        UINT imask；        Interrupt mask
}T_DOVR;
```

Although NORTi does not refer the value of avcatr, in order to maintain the compatibility with the μITRON of other companies, Please set avcatr to TA_HLNG, which shows that the handler is described with the high-level language. The value of exinf is passed as the second argument of an overrun handler.

ovrhdr is a pointer to the function used as an overrun handler. Please describe an overrun handler as a void type function as follows.

```
void ovrhdr(ID tskid, VP_INT exinf)
{
        :
        :
}
```

Please specify the periodic interruption number, which an overrun handler uses as intno. Generally, the periodic interruption number same as a system clock is used. Please specify the function for clearing an interrupt as ovrclr. When the interrupt number of a system clock is used, please specify NULL as ovrclr.

In order to use a different interrupt number from a system clock, it is necessary to create the unique initialization routine and ovrclr function. The function registered into ovrclr is called whenever interruption enters.

If NULL is specified as pk_dovr, an overrun handler definition will be canceled. An overrun handler will be redefined if values other than NULL are again specified as pk_dovr in the state of already giving the definition. When you use peculier interrupts, please cancel the definition / redefine after forbidding interrupt.

Return	E_OK	Successful termination
	E_NOID	The interrupt service routine ID is Insufficient
	E_CTX	Issued from an interrupt handler *
	E_SYS	The memory for a management block is not securable. **
	E_PAR	Interrupt number intno is outside range *
	Others	Error code of acre_isr if pk_dovr = NULL
		Error code of del_isr if pk_dovr ≠ NULL

Example

```
#define INT_CMT INT_CMI0
extern void ovrhdr(ID, VP_INT);
const T_DOVR dovr ={TA_HLNG, ovrhdr, INT_CMT, NULL,0x07};

TASK task1(void)
{
    ER ercd;
        :
    ercd = def_ovr(&dovr);
        :
}
```

sta_ovr

Function	Start operation of overrun handler

Declaration ER sta_ovr(ID tskid, OVRTIM ovrtim);

tskid	ID of the task which sets up time
ovrtim	Overrun time

Description Processor time is set up by the task specified by tskid. It will be aimed at a self-task if TSK_SELF is specified as tskid. Time unit is the interrupt cycle specified by def_ovr. An overrun handler will be started if the time specified by ovrtim is used up.
Measuring of processor time is done by decrementing the processor time of the task, which was performed at the time of interrupt, by 1. Hence, continued execution time, except in the case of a very long task of a interrupt cycle, becomes the large error.

Processor time will be updated if sta_ovr is again performed by the task to which processor time is already set.

Return

E_OK	Successful termination
E_ID	Task ID is invalid *
E_NOEXS	Task do not exist
E_PAR	Incorrect time
E_OBJ	Overrun handler is not definied

stp_ovr

Function	Stop Overrun handler operation

Declaration ER stp_ovr(ID tskid);

tskid	ID of the task which suspends a time check

Description Operation of an overrun handler is stopped by the task specified by tskid. A setup of processor time is canceled. A self-task can be specified by tskid = TSK_SELF.

Return

E_OK	Successful termination
E_ID	Task ID is invalid *
E_OBJ	Overrun handler is not defined

ref_ovr

Function Refer to overrun handler state.

Declaration ER ref_ovr(ID tskid, T_ROVR *pk_rovr);

tskid	ID of the task which refers to processor time
pk_rovr	The pointer to the location which stores an overrun handler state packet

Description The state of the overrun handler of the task specified by tskid is returned to *pk_rovr. A self-task can be specified by tskid = TSK_SELF.

The structure of an overrun handler state packet is as follows.

```
typedef struct t_rovr
{       STAT ovrstat;         The state of an overrun handler
        OVRTIM leftotm;       The processor remaining time
}T_ROVR;
```

The following value returns to ovrstat.

TOVR_STP	Processor time is not set up.
TOVR_STA	Processor time is set up.

The remaining time to starting returns to leftotm.

Return

E_OK	Successful termination
E_ID	Incorrect task ID *
E_OBJ	Overrun handler not definied

Example

```
TASK task1(void)
{
        T_ROVR rovr;
                :
        ref_ovr(TSK_SELF, &rovr);
        if(rovr.leftotm > 100/MSEC)
                :
}
```

5.15 Service call management functions

def_svc

Function A definition of an extended service call

Declaration ER def_svc(FN fncd, const T_DSVC *pk_dsvc);

fncd	Functional code of the definition
pk_dsvc	The pointer to the packet which stores extended service call definition information

Description pk_dsvc defines the extended service call specified by fncd.

The structure of an extended service call definition information packet is as follows.

```
typedef struct t_dsvc
{       ATR svcatr;           Extended service call attribute
        FP svcrtn;            Extended service call routine address
        INT parn;             The number of parameters of an extended service call routine
}T_DSVC;
```

Please set a positive value to fncd. Although NORTi does not refer the value of avcatr, in order to maintain the compatibility with the μITRON of other companies, Please set avcatr to TA_HLNG, which shows that the handler is described with the high-level language.

Please describe an extended service call routine as a C function in the following form.

```
ER_UINT svcrtn(VP_INT par1, VP_INT par2 ,..., VP_INT par6)
{
        :
        :
}
```

Please set the number of parameters to parn. Number of parameters can be minumum 0 and maximum 6. An extended service call routine is a subroutine performed in the called context. It is also possible to register a standard system call as an extended service call.

Return

E_OK	Successful termination
E_CTX	Issued from a non-task context *
E_PAR	Parameter error

Example

```
#define svc_ter_tsk    2
const T_DSVC dsvc2 = {TA_HLNG, (FP)v4_ter_tsk, 1};

TASK task1(void)
{
            :
    ercd = def_svc(svc_ter_tsk, &dsvc2);
            :
}
```

cal_svc

Function	A call of a service call

Declaration ER_UINT cal_svc(FN fncd, VP_INT par1, VP_INT par2, ...);

fncd	The service call functional code
par1	The first parameter passed to a service call routine
par2	The second parameter passed to a service call routine
	...
par6	The sixth parameter passed to a service call routine

Description par1-par6 are called for the service call routine specified by fncd as a parameter. A parameter should describe only a required number.

Return The return value from a service call

E_RSFN	Service call routine undefined
E_PAR	incorrect fncd *

Example

```
#define svc_ter_tsk    2
#define Task2 2
const T_DSVC dsvc2 = {TA_HLNG, (FP)v4_ter_tsk, 1};

TASK task1(void)
{
            :
    ercd = def_svc(svc_ter_tsk, &dsvc2);
            :
    ercd = cal_svc(svc_ter_tsk, Task2);
            :
}
```

5.16 System state management functions

rot_rdq
irot_rdq

Function Task ready queue rotation

Declaration ER rot_rdq(PRI tskpri);
ER irot_rdq(PRI tskpri);
tskpri Priority

Description In the ready queue of the priority specified by tskpri, the task at the head position is switched to the tail end. That is, execution of the task of the same priority is switched.
By tskpri = TPRI_SELF, the base priority of a self-task is made into an target priority. By using this system call at a fixed interval from a cyclic handler, a round Robins scheduling is realizable.
When the ready queue of the task which published this system call rotates, this task is transited from a RUNNING state to a ready state, and the task which was waiting for an execution order next transits it from a ready state to a RUNNING state. That is, rot_rdq can be published in order to abandon the right of execution itself.
There is no error in case this system call is issued when there is no task in the ready queue of the specified priority.

Return E_OK Successful termination
E_PAR Priority is out of range *

Example

```
TASK task1(void)
{
        :
    rot_rdq(TPRI_SELF);
        :
}
```

get_tid
iget_tid

Function Refer to task ID of an execution task.

Declaration ER get_tid(ID *p_tskid);

ER iget_tid(ID *p_tskid);

p_tskid The pointer to the location which stores Task ID

Description The ID number of the task which issued this system call is returned to *p_tskid. When called from the non-task context sections, such as an interrupt handler, ID of the task in a present RUNNING state is returned. TSK_NONE is returned when there is no task with a RUNNING state.

Return E_OK Successful termination

Example

```
TASK task1(void)
{
    ID tskid;
        :
    get_tid(&tskid);
        :
}
```

vget_tid

Function Get the task ID of the self-task.

Declaration ID vget_tid(void);

Description The ID number of the task, which issued this system call, is returned as a function return value. Others are the same as that of get_tid.

Return Task ID

Note This system call is unique to NORTi

Example

```
TASK task1(void)
{
        ID tskid;
                :
        tskid = vget_tid();
                :
}
```

loc_cpu
iloc_cpu

Function Change to CPU locked state (Disables interrupt and dispatch)

Declaration ER loc_cpu(void);
ER iloc_cpu(void);

Description A reception of interrupt and task switching are prohibited. This prohibition state can be canceled by the unl_cpu system call. If this system call is issued when it is already in a CPU lock state, it does not become an error.
However, since the nest management of loc_cpu~unl_cpu pair is not done, CPU lock release will be done by single unl_cpu call, even if loc_cpu was issued multiple times.
Please do not publish this system call from an interrupt handler. In case when CPU lock command is issued from non-task context other than interrupt handler, please release the CPU lock state before return.

Return E_OK Successful termination

Note In the case of a processor with a level interrupt function, in NORTi, as a standard, the interrupt inhibit level of the Kernel is not considered as highest. The interrupt mask set up by loc_cpu, disables even the interrupt-inhibit level of a kernel. The interrupts with priority higher than Kernel can be receieved.

unl_cpu
iunl_cpu

Function Release of a CPU lock state

Declaration ER unl_cpu(void);
ER iunl_cpu(void);

Description The prohibition state set up by loc_cpu is canceled. However, interrupt reception and task switching are not necessarily enabled. When loc_cpu was issued while dispatch was prohibited, dispatch remains prohibited when CPU is unlocked. In this case, in order to make dispatch possible, ena_dsp should be called.
When already in CPU lock released state, repeated use of this system call does not become an error. Since the nest management of loc_cpu~unl_cpu pair is not done, CPU lock release will be done by single unl_cpu call, even if loc_cpu was issued multiple times.
Although it is possible to call iunl_cpu from a timer event handler among non-task contexts, please do not publish this system call from an interrupt handler. All interrupt masks will be canceled. In case of the processor that supports level interrupt; when unl_loc is called at the time of return from ent_int in the interrupt handler (interrupt service routine in case of Interrupt Service Routine), the interrupt mask is cleared.

Return E_OK Successful termination

dis_dsp

Function Disable dispatch

Declaration ER dis_dsp(void);

Description The task switching is forbidden. Interrupt is not forbidden. After issuing this system call, switching of tasks issued by other system calls is suspended. The switching of the suspended task is performed when an ena_dsp system call is issued.

Notes During the bans on dispatch, if the wait generating system call is issued, it will become an E_CTX error.

Return

E_OK	Successful termination
E_CTX	Issue from the non-task context section *

Example

```
TASK task1(void)
{
        :
    dis_dsp();
        :       /* Dispatch prohibited */
    ena_dsp();
        :
}
```

ena_dsp

Function	Dispatch permission
Declaration	ER ena_dsp(void);
Description	The dispatch prohibition state set up by the dis_dsp system call is canceled. Even if dis_dsp is called previously, it is not considered as an error. If there is a switching of the task suspended in the state of dispatch prohibition, it will perform by this system call.

Return	E_OK	Successful termination
	E_CTX	Issued from a non-task context *

sns_ctx

Function	Refer to context.
Declaration	BOOL sns_ctx(void);
Description	It is TRUE when called from the non-task context section. FALSE is returned when called from the task context section.

Return	TRUE	Non-task context
	FALSE	Task context section

sns_loc

Function Refer to CPU lock state.

Declaration BOOL sns_loc(void);

Description TRUE is returned in case the CPU is in locked state. In other case FALSE is returned.

Return

TRUE	CPU is locked.
FALSE	CPU is unlocked.

Example

```
BOOL cpu_lock = sns_loc();
        :
if(!cpu_lock)
        loc_cpu();
        :
/* Processing while CPU is in locked state */
        :
if(!cpu_lock)              /* In order not to carry out lock release carelessly */
        unl_cpu();
        :
```

sns_dsp

Function Refer to dispatch prohibition state.

Declaration BOOL sns_dsp(void);

Description TRUE is returned if the system is in dispatch prohiition state. When the dispatch is permitted, FALSE is returned.

Return

TRUE	Dispatch prohibition state
FALSE	Dispatch permission state

Example

```
BOOL task_lock = sns_dsp();
        :
if (!task_lock)
        dis_dsp();
        :
/* Processing at the time of dispatch prohibition state */
        :
if (!task_lock)          /* In order not to carry out dispatch permission incorrectly */
        ena_dsp();
        :
```

sns_dpn

Function Refer to dispatch suspension state.

Declaration BOOL sns_dpn(void);

Description TRUE is returned if the CPU is in locked state or dispatch is banned. In other case FALSE is returned.

Return

TRUE	Dispatch suspension state
FALSE	Dispatch is not banned

ref_sys

Function Refer to system state.

Declaration ER ref_sys(T_RSYS *pk_rsys);

pk_rsys The pointer to the location which stores a system state packet

Description The running state of OS is returned to *pk_rsys.

The structure of a system state packet is as follows.

```
typedef struct t_rsys
{
    INT sysstat;        System state
}T_RSYS;
```

The any of following values is returned to sysstat.

TSS_TSK	The task context section is under execution and dispatch is permitted.
TSS_DDSP	The task context section is under execution and dispatch is forbidden.
TSS_LOC	The task context section is under execution and interrupt, dispatch is forbidden.
TSS_INDP	The non-task context section is under execution.

Return E_OK Successful termination

Example

```
TASK task1(void)
{
    T_RSYS rsys;
        :
    ref_sys(&rsys);
    if(rsys.sysstat == TSS_LOC)
        :
}
```

5.17 System configuration management functions

ref_ver

Function Version reference

Declaration ER ref_ver(T_RVER *pk_rver);

pk_rver The pointer to the location which stores a version information packet

Description The version of NORTi is returned to *pk_ver.

The structure of a version information packet is as follows.

```
typedef struct t_rver
{   UH maker;      Maker (0108H: MiSPO Co., Ltd.)
    UH prid;       format number
    UH spver;      Specification version
    UH prver;      Product version
    UH prno[4];    Product management information
}T_RVER;
```

Please refer to μITRON specification about the detailed meaning of the member of a structure object. Refer to source file n4cxxx.asm of a kernel about the value actually returned.

Return E_OK Successful termination

ref_cfg

Function Refer to configuration information.

Declaration ER ref_cfg(T_RCFG *pk_rcfg);

pk_rcfg	The pointer to the location which stores a configuration information packet

Description Configuration information is returned to *pk_rcfg.

The structures of configuration information packets are unique to NORTi.

```
typedef struct t_rcfg
{       ID tskid_max;       Task ID maximum
        ID semid_max;       Semaphore ID maximum
        ID flgid_max;       Event flag ID maximum
        ID mbxid_max;       Mail box ID maximum
        ID mbfid_max;       Message buffer ID maximum
        ID porid_max;       The rendezvous port ID maximum
        ID mplid_max;       Variable-length memory pool ID maximum
        ID mpfid_max;       Fixed-length memory pool ID maximum
        ID cycno_max;       Cyclic handler ID maximum
        ID almno_max;       Alarm handler ID maximum
        PRI tpri_max;       Task priority maximum
        int tmrqsz;         Timer queue size of a task (the number of bytes)
        int cycqsz;         Timer queue size of a cyclic handler (the number of bytes)
        int almqsz;         Timer queue size of an alarm handler (the number of bytes)
        int istksz;         Stack size of an interrupt handler (the number of bytes)
        int tstksz;         Stack size of a time event handler (the number of bytes)
        SIZE sysmsz;        Size of system memory (the number of bytes)
        SIZE mplmsz;        Size of the memory for a memory pool (the number of bytes)
        SIZE stkmsz;        Size of the memory for stacks (the number of bytes)
        ID dtqid_max;       Data queue ID maximum
        ID mtxid_max;       Mutex ID maximum
        ID isrid_max;       Interrupt-service-routine ID maximum
        ID svcfn_max;       Extended service call functional number maximum
        :(more may be added in the future)
}T_RCFG;
```

Return

E_OK	Successful termination

6. Exclusive System Calls

6.1 NORTi Exclusive System management functions

sysini

Function	System Initialization
Declaration	ER sysini(void);

Description

The sysini system call initializes the kernel. This system call must be executed before all other system calls. It is usually called at the top of main functions.

The initialization process executed in this case is the initial setting of internal kernel variables and the calling of intini functions stated later. After the sysini system call is executed, the process enters the interrupt-disabled state.

When the standard stack area that the compiler offers is used as a stack memory, that is, configuration #define STKMSZ 0, the bottom of the stack will be allocated, based on a stack pointer at the time of call to sysini.

When the configurator is used, it is automatically called from the main function generated by configurator (kernel_cfg.c).

Return

E_OK	Successful termination
E_SYS	Insufficient memory for management block **
E_NOMEM	Insufficient memory for stack **
Others	return values from intini function.

syssta

Function	Start the system
Declaration	ER syssta(void);

Description

The syssta system call transfers the system to the multi-task state, terminating the handler for initialization. At least more than one task's creation and start have to be executed before this system call is issued. This system call is usually called at the end of the main functions.

In activated tasks, the task with the highest priority has control (for tasks with the same priority, the task activated earlier) i.e. the first dispatch is executed. After this, the interrupts, which were prohibited by sysini, are permitted.

When the error has occurred in task generation etc. before syssta execution, an error returns without system start. The syssta call does not return in normal start.

When the configurator is used, it is automatically called from the main function generated by configurator (kernel_cfg.c).

Return

E_PAR	The priority, etc. are out of the range. *
E_ID	The ID is out of the range. *
E_OBJ	Already created.
E_SYS	Memory shortage for a management block. **
E_NOMEM	Memory shortage for stack and memory pool **

intsta

Function Start periodic timer interrupt

Declaration ER intsta(void);

Description Periodic timer interrupt for managing the time waiting of a task is started. Please call this function just before a syssta system call. It is not necessary to perform intsta when not using a system call or a timer event handler with a timeout.

As this system call depends on the target, it is defined in n4ixxx.c, different from the kernel. Standard value for interrupt cycle is 10msec. User needs to create this function if it is not defined in sample n4ixxx.c file. In such case user may change the function name. When configurator is used, it is called automatically from main function defined in configurator (kernel_cfg.c).

When you use periodic timer interrupt from the overrun handler, please call def_ovr after periodic timer interrupt initialization.

Return

E_OK	Successful termination
E_PAR	The interrupt vector size is out of the range (depending on the target).

intext

Function Terminate periodic timer interrupt

Declaration void intext();

Description The intext system call stops the timer activated by intsta.

As this system call depends on the target, it is defined in n4ixxx.c, different from the kernel. Please create this function if the attched n4ixxx.c file does not include this function. When user defines this function, the name of this function can be changed. User need not define this function if there is no need to stop the timer interrupt. (It is omitted in many of the samples)

Return none

intini

Function	Interrupt Initialization
Declaration	ER intini(void);
Description	The intini system call is called in the interrupt-disabled state from sysini. It initializes the hardware, and so on. As this system call depends on the target, it is defined in the attached n4ixxx.c, which is supplied as a sample, different from the kernel. When a user creates this function, if there is nothing specially to initialize, please do nothing but carry out the return with E_OK code.

Return	E_OK	Successful termination
	E_PAR	The interrupt vector size is out of the range (depending on the target).

7. List

7.1 Error code list

E_OK	0	Normal termination / Successful termination
E_SYS	0xf..ffb (-5)	System error
E_NOSPT	0xf..ff7 (-9)	Unsupported function
E_RSFN	0xf..ff6 (-10)	Subscription/reservation function code
E_RSATR	0xf..ff5 (-11)	Subscription attribute
E_PAR	0xf..fef (-17)	Parameter error
E_ID	0xf..fee (-18)	Illegal ID number
E_CTX	0xf..fe7 (-25)	Context error
E_ILUSE	0xf..fe4 (-28)	Illegal use of system call
E_NOMEM	0xf..fdf (-33)	Insufficient memory
E_NOID	0xf..fde (-34)	Insufficient ID number
E_OBJ	0xf..fd7 (-41)	Object function error
E_NOEXS	0xf..fd6 (-42)	Uncreated object
E_QOVR	0xf..fd5 (-43)	Queuing overflow
E_TMOUT	0xf..fce (-50)	Polling failure or timeout
E_RLWAI	0xf..fcf (-49)	Forced release of wait state
E_DLT	0xf..fcd (-51)	Deletion of waiting object

7.2 System call list

Task management functions

	1 2 3
Task creation cre_tsk (tskid, pk_ctsk) ;	O O X
Task creation (Automatic ID allocation) acre_tsk (pk_ctsk) ;	O O X
Task Deletion del_tsk(tskid);	O O X
Task activation act_tsk(tskid);	O O O
Task starting iact_tsk(tskid);	X O O
Cacellation of task start command can_act(tskid);	O O O
Task starting (Starting code specification) sta_tsk(tskid, stacd);	O O O
Self-task termination ext_tsk();	O X X
Self-task terminmation and deletion exd_tsk();	O X X
Other task forced termination ter_tsk(tskid);	O O O
Change task priority chg_pri(tskid, tskpri);	O O O
Refer to task priority get_pri(tskid, p_tskpri);	O O O
Refer to task state ref_tsk(tskid, pk_rtsk);	O O O
Refer to task state (Simple version) ref_tst(tskid, pk_rtst);	O O O

Notes,
1 – Can Issue from task.
2 – Can issue from timer /event handler.
3 – Can issue from interrupt handler.

Task associated synchronization

	1	2	3
Waiting for wakeup slp_tsk();	O	X	X
Waiting for wakeup (timeout specified) tslp_tsk(tmout);	O	X	X
Task wakeup command wup_tsk(tskid);	O	O	O
Task wakeup command iwup_tsk(tskid);	X	O	O
Cancellation of task wakeup command can_wup(tskid);	O	O	O
Self-task wakeup command cancellation * vcan_wup(tskid);	O	O	O
Forced release of waiting task rel_wai(tskid);	O	O	O
Forced release of waiting task irel_wai(tskid);	X	O	O
Task suspend command sus_tsk(tskid);	O	O	O
Resume from suspended state rsm_tsk(tskid);	O	O	O
Forced resume from suspended state frsm_tsk(tskid);	O	O	O
Delay self-task dly_tsk(dlytim);	O	X	X

Notes,
* NORTi original system call
1 – Can Issue from task.
2 – Can issue from timer /event handler.
3 – Can issue from interrupt handler.

Task exception handling

	1 2 3
Definition of the task exception handling routine def_tex(tskid, pk_dtex);	O O X
Request task exception handling ras_tex(tskid, rasptn);	O O O
Request task exception handling iras_tex(tskid, rasptn);	X O O
Prohibit task exception handling dis_tex();	O O O
Enable task exception handling ena_tex();	O O O
Refer to task exception handling prohibition state sns_tex();	O O O
Refer to the state of the task exception handling ref_tex(tskid, pk_rtex);	O O O

Notes,
1 – Can Issue from task.
2 – Can issue from timer /event handler.
3 – Can issue from interrupt handler.

Synchronization and Communication (Semaphore)

	1 2 3
Semaphore creation cre_sem(semid, pk_csem);	O O X
Semaphore creation (Automatic ID allocation) acre_sem(pk_csem);	O O X
Semaphore deletion del_sem(semid);	O O X
Semaphore resource release sig_sem(semid);	O O O
Semaphore resource release isig_sem(semid);	X O O
Semaphore resource acquisition wai_sem(semid);	O X X
Semaphore resource acquisition (polling) pol_sem(semid);	O O O
Semaphore resource acquisition (timeout available) twai_sem(semid, tmout);	O X X
Semaphore state reference ref_sem(semid, pk_rsem);	O O O

Notes,
1 – Can Issue from task.
2 – Can issue from timer /event handler.
3 – Can issue from interrupt handler.

Synchronization and Communication (Event flag)

	1 2 3
Event flag creation cre_flg(flgid, pk_cflg);	O O X
Event flag creation (automatic ID allocation) acre_flg(pk_cflg);	O O X
Event flag deletion del_flg(flgid);	O O X
Event flag set set_flg(flgid, setptn);	O O O
Event flag set iset_flg(flgid, setptn);	X O O
Event flag clear clr_flg(flgid, clrptn);	O O O
Waiting for event flag wai_flg(flgid, waiptn, wfmode, p_flgptn);	O X X
Waiting for event flag (polling mode) pol_flg(flgid, waiptn, wfmode, p_flgptn);	O O O
Waiting for event flag (timeout available) twai_flg(flgid, waiptn, wfmode,p_flgptn, tmout);	O X X
Refer to event flag state ref_flg(flgid, pk_rflg);	O O O

Notes,
1 – Can Issue from task.
2 – Can issue from timer /event handler.
3 – Can issue from interrupt handler.

Synchronization and Communication (Data queue)

	1 2 3
Data queue creation cre_dtq(dtqid, pk_cdtq);	O O X
Data queue creation (automatic ID allocation) acre_dtq(pk_cdtq);	O O X
Data queue deletion del_dtq(dtqid);	O O X
Send data queue snd_dtq(dtqid, data);	O X X
Send data queue (polling mode) psnd_dtq(dtqid, data);	O O O
Send data queue (polling mode) ipsnd_dtq(dtqid, data);	X O O
Send data queue (timeout available) tsnd_dtq(dtqid, data, tmout);	O X X
Forced transmission to data queue fsnd_dtq(dtqid, data);	O O O
Forced transmission to data queue ifsnd_dtq(dtqid, data);	X O O
Reception of data queue rcv_dtq(dtqid, p_data);	O X X
Reception of data queue (polling mode) prcv_dtq(dtqid, p_data);	O O O
Reception of data queue (timeout available) trcv_dtq(dtqid, p_data, tmout);	O X X
Refer to the state of data queue ref_dtq(dtqid, pk_rdtq);	O O O

Notes,
1 – Can Issue from task.
2 – Can issue from timer /event handler.
3 – Can issue from interrupt handler.

Synchronization and Communication (Mail box)

	1 2 3
Mailbox creation cre_mbx(mbxid, pk_cmbx);	O O X
Mailbox creation (automatic ID allocation) acre_mbx(pk_cmbx);	O O X
Mailbox deletion del_mbx(mbxid);	O O X
Send message to mailbox snd_mbx(mbxid, pk_msg);	O O O
Receive message from mailbox rcv_mbx(mbxid, ppk_msg);	O X X
Receive message from mailbox (polling mode) prcv_mbx(mbxid, ppk_msg);	O O O
Receive message from mailbox (timeout available) trcv_mbx(mbxid, ppk_msg, tmout);	O X X
Refer to the state of the mailbox ref_mbx(mbxid, pk_rmbx);	O O O

Notes,
1 – Can Issue from task.
2 – Can issue from timer /event handler.
3 – Can issue from interrupt handler.

Extended Synchronization and Communication (Mutex)

	1 2 3
Mutex creation cre_mtx(mtxid, pk_cmtx);	O O X
Mutex creation (automatic ID allocation) acre_mtx(pk_cmtx);	O O X
Mutex deletion del_mtx(mtxid);	O O X
Lock the mutex loc_mtx(mtxid);	O X X
Lock the mutex (polling mode) ploc_mtx(mtxid);	O O O
Lock the mutex (timeout available) tloc_mtx(mtxid,tmout);	O X X
Unlock the mutex unl_mtx(mtxid);	O O O
Refer to the state of the mutex ref_mtx(mtxid, pk_rmtx);	O O O

Notes,
1 – Can Issue from task.
2 – Can issue from timer /event handler.
3 – Can issue from interrupt handler.

Extended Synchronization and Communication (Message buffer)

	1 2 3
Message buffer creation cre_mbf(mbfid, pk_cmbf);	O O X
Message buffer creation (automatic ID allocation) acre_mbf(pk_cmbf);	O O X
Message buffer deletion del_mbf(mbfid);	O O X
Send message to message buffer. Snd_mbf(mbfid, msg, msgsz);	O X X
Send message to message buffer (polling mode) psnd_mbf(mbfid, msg, msgsz);	O O O
Send message to message buffer (timeout available) tsnd_mbf(mbfid, msg, msgsz, tmout);	O X X
Receive message from message buffer. Rcv_mbf(mbfid, msg);	O X X
Receive message from message buffer (polling mode) prcv_mbf(mbfid, msg);	O O O
Receive message from message buffer (timeout available) trcv_mbf(mbfid, msg, tmout);	O X X
Refer to the state of the message buffer ref_mbf(mbfid, pk_rmbf);	O O O

Notes,
1 – Can Issue from task.
2 – Can issue from timer /event handler.
3 – Can issue from interrupt handler.

Extended Synchronization and Communication (Rendezvous port)

	1 2 3
Rendezvous port creation cre_por(porid, pk_cpor);	O O X
Rendezvous port creation (automatic ID allocation) acre_por(pk_cpor);	O O X
Rendezvous port deletion del_por(porid);	O O X
Call Rendezvous port cal_por(porid, calptn, msg, cmsgsz);	O X X
Call Rendezvous port (timeout available) tcal_por(porid, calptn, msg, cmsgsz, tmout);	O X X
Waiting for rendezvous port acp_por(porid, acpptn, p_rdvno, msg);	O X X
Waiting for rendezvous port (polling mode) pacp_por(porid, acpptn, p_rdvno, msg);	O O O
Waiting for rendezvous port (timeout available) tacp_por(porid, acpptn, p_rdvno, msg, tmout);	O X X
Transfer of rendezvous fwd_por(porid, calptn, rdvno, msg, cmsgsz);	O O O
End of rendezvous rpl_rdv(rdvno, msg, rmsgsz);	O O O
Refer to the state of rendezvous port. Ref_por(porid, pk_rpor);	O O O
Refer to the state of rendezvous. Ref_rdv(rdvno, pk_rrdv);	O O O

Notes,
1 – Can Issue from task.
2 – Can issue from timer /event handler.
3 – Can issue from interrupt handler.

Fixed length memory pool management

	1 2 3
Fixed-length memory pool creation cre_mpf(mpfid, pk_cmpf);	O O X
Fixed-length memory pool creation (automatic ID allocation) acre_mpf(pk_cmpf);	O O X
Fixed-length memory pool deletion del_mpf(mpfid);	O O X
Fixed-length memory block acquisition get_mpf(mpfid, p_blk);	O X X
Fixed-length memory block acquisition (polling) pget_mpf(mpfid, p_blk);	O O O
Fixed-length memory block acquisition (timeout) tget_mpf(mpfid, p_blk, tmout);	O X X
Fixed-length memory block release rel_mpf(mpfid, blk);	O O O
Refer to the state of the fixed size memory pool. Ref_mpf(mpfid, pk_rmpf);	O O O

Notes,
1 – Can Issue from task.
2 – Can issue from timer /event handler.
3 – Can issue from interrupt handler.

Variable length memory pool management

	1 2 3
Variable-length memory pool creation cre_mpl(mplid, pk_cmpl);	O O X
Variable-length memory pool creation (automatic ID allocation) acre_mpl(pk_cmpl);	O O X
Variable-length memory pool delation del_mpl(mplid);	O O X
Acquisition of block from variable-length memory pool. Get_mpl(mplid, blksz, p_blk);	O X X
Acquisition of block from variable-length memory pool (polling mode) pget_mpl(mplid, blksz, p_blk);	O O X
Acquisition of block from variable-length memory pool (timeout available) tget_mpl(mplid, blksz, p_blk, tmout);	O X X
Variable-length memory pool release rel_mpl(mplid, blk);	O O X
Refer to the state of the variable length memory pool ref_mpl(mplid, pk_rmpl);	O O X

Notes,
1 – Can Issue from task.
2 – Can issue from timer /event handler.
3 – Can issue from interrupt handler.

Time management (System time)

	1	2	3
A setup of the system time set_tim(p_tim);	O	O	O
Get the system time get_tim(p_tim);	O	O	O
Supply of a time tick isig_tim();	X	X	O
Supply of a time tick sig_tim();	X	X	O

Notes,
1 – Can Issue from task.
2 – Can issue from timer /event handler.
3 – Can issue from interrupt handler.

Time management (Cyclic handler)

	1 2 3
Cyclic handler creation cre_cyc(cycid, pk_ccyc);	O O X
Cyclic handler creation (automatic ID allocation) acre_cyc(pk_ccyc);	O O X
Cyclic handler deletion del_cyc(cycid);	O O X
Start the cyclic handler sta_cyc(cycid);	O O O
Stop the cyclic handler stp_cyc(cycid);	O O O
Refer to the state of the cyclic handler ref_cyc(cycid, pk_rcyc);	O O O

Notes,
1 – Can Issue from task.
2 – Can issue from timer /event handler.
3 – Can issue from interrupt handler.

Time management (Alarm handler)

	1 2 3
Alarm handler creation cre_alm(almid, pk_calm);	O O X
Alarm handler creation (automatic ID allocation) acre_alm(pk_calm);	O O X
Alarm handler deletion del_alm(almid);	O O X
Start of the alarm handler sta_alm(almid, almtim);	O O O
Stop the alarm handler stp_alm(almid);	O O O
Refer to the state of the alarm handler ref_alm(almid, pk_ralm);	O O O

Notes,
1 – Can Issue from task.
2 – Can issue from timer /event handler.
3 – Can issue from interrupt handler.

Time management (Overrun handler)

	1	2	3
Overrun handler definition def_ovr(pk_dovr);	O	O	X
Start of the overrun handler sta_ovr(tskid, ovrtim);	O	O	O
Stop the overrun handler stp_ovr(tskid);	O	O	O
Refer to the state of the overrun handler ref_ovr(tskid, pk_rovr);	O	O	O

Notes,
1 – Can Issue from task.
2 – Can issue from timer /event handler.
3 – Can issue from interrupt handler.

System state management

	1 2 3
Rotation of the task execution order. Rot_rdq(tskpri);	O O O
Rotation of the task execution order. Irot_rdq(tskpri);	X O O
Refer to the task ID of a running state get_tid(p_tskid);	O O O
Refer to the task ID of a running state iget_tid(p_tskid);	X O O
Refer to the state of the self-task * vget_tid();	O O O
Set CPU to lock state loc_cpu();	O O X
Set CPU to lock state iloc_cpu();	X O X
Unlock the CPU locked state unl_cpu();	O O X
Unlock the CPU locked state iunl_cpu();	X O X
Prohibit the dispatch dis_dsp();	O X X
Enable the dispatch ena_dsp();	O X X
Refer to the state of the system ref_sys(pk_rsys);	O X X
Refer to the context sns_ctx();	O O O
Refer to the CPU lock state sns_loc();	O O O
Refer to the dispatch prohibition state sns_dsp();	O O O
Refer to the dispatch suspension state sns_dpn();	O O O

Notes,
* System call exclusive to NORTi
1 – Can Issue from task.
2 – Can issue from timer /event handler.
3 – Can issue from interrupt handler.

Interrupt management

	1 2 3
Definition of the interrupt handler def_inh(inhno, pk_dinh);	O O O
Interrupt service routine creation cre_isr(isrid, pk_cisr);	O O X
Interrupt service routine creation (automatic ID allocation) acre_isr(pk_cisr);	O O X
Interrupt service routine deletion del_isr(isrid);	O O X
Refer to the interrupt service routine state. Ref_isr(isrid, pk_risr);	O O O
Prohibition of the interrupt. Dis_int(intno);	O O X
Enable the interrupt. Ena_int(intno);	O O X
Change of the interrupt mask. Chg_ims(imask);	O O O
Get the interrupt mask state. Get_ims(p_imask);	O O O
Start the interrupt handler * ient_int();	X X O
End the interrupt handler * iret_int();	X X O
Set the status register * vset_psw();	O O O
Set the interrupt mask state of the status register. * vdis_psw();	O O O

Notes,
* System call exclusive to NORTi
1 – Can Issue from task.
2 – Can issue from timer /event handler.
3 – Can issue from interrupt handler.

Service call management functions

	1 2 3
Extended service call definition def_svc(fncd, pk_dsvc);	O O X
Calling the service call. Cal_svc(fncd, par1, par2, …);	O ? ? (depend on the service call)

Notes,
1 – Can Issue from task.
2 – Can issue from timer /event handler.
3 – Can issue from interrupt handler.

System configuration management

	1 2 3
Refer to configuration information. Ref_cfg(pk_rcfg);	O O O
Refer to version information. Ref_ver(pk_rver);	O O O

Notes,
1 – Can Issue from task.
2 – Can issue from timer /event handler.
3 – Can issue from interrupt handler.

7.3 Static API list

(The content of this section is moved to NORTi Configurator manual book.)

7.4 Packet structure object list

Task generation information packet

```
typedef struct t_ctsk
{   ATR tskatr;        Task attribute
    VP_INT exinf;      Task extension information
    FP task;           Pointer to the function made as a task
    PRI itskpri;       Task priority at start
    SIZE stksz;        Stack size (number of bytes)
    VP stk;            Stack domain start address
    B *name;           the pointer to task name
}T_CTSK;
```

Task state packet

```
typedef struct t_rtsk
{   STAT tskstat;      Task state
    PRI tskpri;        Current priority of task
    PRI tskbpri;       Base priority
    STAT tskwait;      Waiting factor
    ID wid;            Waiting object ID
    TMO lefttmo;       remaining value of timeout time
    UINT actcnt;       Startup request count
    UINT wupcnt;       Wakeup request count
    UINT suscnt;       Suspend demand count
    VP exinf;          Extended information
    ATR tskatr;        task attribute
    FP task;           pointer to task function
    PRI itskpri;       task priority at the time of starting
    SIZE stksz;        Stack size (in bytes)
}T_RTSK;
```

Task state easy reference packet

```
typedef struct t_rtst
{   STAT tskstat;      Task state
    STAT tskwait;      Waiting factor
}T_RTST;
```

Task exception handler generation information packet

```
typedef struct t_dtex
{   ATR texatr;        Task exception handler attribute
    FP texrtn;         Pointer to task exception handler function
}T_DTEX;
```

Task exception handler state packet

```
typedef struct t_rtex
{       STAT texstat;         Task exception processing state
        TEXPTN pndptn;        Pending exception code
}T_RTEX;
```

Semaphore generation information packet

```
typedef struct t_csem
{       ATR sematr;           Semaphore attribute
        UINT isemcnt;         Semaphore initial count
        UINT maxsem;          Semaphore maximum count
        B *name;              pointer to the semaphore name
}T_CSEM;
```

Semaphore state packet

```
typedef struct t_rsem
{       ID wtskid;            Waiting task ID
        UINT semcnt;          Semaphore count
}T_RSEM;
```

Event flag generation information packet

```
typedef struct t_cflg
{       ATR flgatr;           Event flag attribute
        FLGPTN iflgptn;       Event flag initial value
        B *name;              pointer to the event flag name
}T_CFLG;
```

Event flag state packet

```
typedef struct t_rflg
{       ID wtskid;            Waiting task ID
        FLGPTN flgptn;        Event flag value
}T_RFLG;
```

Data queue generation information packet

```
typedef struct t_cdtq
{       ATR dtqatr;           Data queue attribute
        UINT dtqcnt;          Data queue size (data size)
        VP dtq;               Rig buffer address
        B *name;              Pointer to the data queue name
}T_CDTQ;
```

Data queue state packet

```
typedef struct t_rdtq
{   ID stskid;          ID of the task waiting for transmission
    ID rtskid;          ID of the task waiting for reception
    UINT sdtqcnt;       Count of data in data-queue
}T_RDTQ;
```

Mailbox generation information packet

```
typedef struct t_cmbx
{   ATR mbxatr;         Mailbox attribute
    PRI maxmpri;        number of message priorities
    VP mprihd;          pointer to message queue header
    B *name;            pointer to the mailbox name
}T_CMBX;
```

Mailbox state packet

```
typedef struct t_rmbx
{   ID wtskid;          reception waiting task ID
    T_MSG *pk_msg;      pointer to the next message to be transmitted
}T_RMBX;
```

Mutex generation information packet

```
typedef struct t_cmtx
{   ATR mtxatr;         Mutex attribute
    PRI ceilpri;        Priority upper limit for the ceiling protocol
    B *name;            pointer to the mutex name
}T_CMTX;
```

Mutex state packet

```
typedef struct t_rmtx
{   ID htskid;          ID of the locked task
    ID wtskid;          ID of the task waitig for release
}T_RMTX;
```

Message buffer generation information packet

```
typedef struct t_cmbf
{   ATR mbfatr;         message buffer attribute
    UINT maxmsz;        maximum length of the message
    SIZE mbfsz;         message buffer size
    VP mbf;             message buffer address
    B *name;            pointer to the message buffer name
}T_CMBF;
```

Message buffer state packet

```
typedef struct t_rmbf
{       ID stskid;          ID of the task waiting for transmission
        ID rtskid;          ID of the task waiting for reception
        UINT smsgcnt;       number of messages included in the message buffer
        SIZE fmbfsz;        buffer empty size (in bytes)
}T_RMBF;
```

The rendezvous port generation information packet

```
typedef struct t_cpor
{       ATR poratr;         Rendezvous port attribute
        UINT maxcmsz;       maximum length of call message
        UINT maxrmsz;       maximum length of reply message
        B *name;            the pointer to the rendezvous port name
}T_CPOR;
```

The rendezvous port state packet

```
typedef struct t_rpor
{       ID ctskid;          Call waiting task ID
        ID atskid;          Reply waiting task ID
}T_RPOR;
```

Rendezvous port state packet

```
typedef struct t_rrdv
{
        ID wtskid;          Rendezvous end waiting task ID
}T_RRDV;
```

Interrupt handler definition information packet

```
typedef struct t_dinh
{       ATR inhatr;         Interrupt handler attribute
        FP inthdr;          Interrupt handler function address
        UINT imask;         Interrupt mask
}T_DINH;
```

Interrupt service routine generation information packet

```
typedef struct t_cisr
{       ATR istatr;         Interrupt service routine attribute
        VP_INT exinf;       Extended information
        INTNO intno;        Interrupt number
        FP isr;             Interrupt service routine address
        UINT imask;         Interrupt mask
}T_CISR;
```

Interrupt service routine state packet

```
typedef struct t_risr
{       INTNO intno;            Interrupt number
        UINT imask ;            Interrupt mask
}T_RISR;
```

Variable length memory pool generation information packet

```
typedef struct t_cmpl
{       ATR mplatr;             Variable-length memory pool attribute
        SIZE mplsz;             Variable-length memory pool size (in bytes)
        VP mpl;                 Variable-length memory pool address
        B *name;                the pointer to a variable-length memory pool name
}T_CMPL;
```

Variable length memory pool state reference packet

```
typedef struct t_rmpl
{       ID wtskid;              ID of the task waiting for acquisition
        SIZE fmplsz;            Total size of free memory (in bytes)
        UINT fblksz;            largest size of continuous block (in bytes)
}T_RMPL;
```

Fixed length memory pool generation information packet

```
typedef struct t_cmpf
{       ATR mpfatr;             Fixed-length memory pool attribute
        UINT blkcnt;            the total number of memory blocks
        UINT blfsz;             Size of a memory block (in bytes)
        VP mpf;                 Memory pool address
        B *name;                the pointer to a fixed-length memory pool name
}T_CMPF;
```

Fixed length memory pool state reference packet

```
typedef struct t_rmpf
{       ID wtskid;              ID of the task waiting for acquisition
        UINT frbcnt;            the number of free blocks
}T_RMPF;
```

Cyclic handler generation information packet

```
typedef struct t_ccyc
{       ATR cycatr;             Cyclic handler attribute
        VP_INT exinf;           Extended information
        FP cychdr;              Address of the cyclic handler function
        RELTIM cyctim;          Interval period
        RELTIM cycphs;          Startup phase
}T_CCYC;
```

Cyclic handler state reference packet

```
typedef struct t_rcyc
{       STAT cycstat;       Cyclic handler operation state
        RELTIM lefttim;     Time left to start
}T_RCYC;
```

Alarm handler generation information packet

```
typedef struct t_calm
{       ATR almatr;         Alarm handler attribute
        VP_INT exinf;       Extended information
        FP almhdr;          Address to an alarm handler function
}T_CALM;
```

Alarm handler state reference packet

```
typedef struct t_ralm
{       STAT almstat;       Alarm handler state
        RELTIM lefttim;     time left to start alarm handler
}T_RALM;
```

Overrun handler generation information packet

```
typedef struct t_dovr
{       ATR ovratr;         Overrun handler attribute
        FP ovrhdr;          Addres to an overrun handler function
        INTNO intno;        interrupt number to be used
        FP ovrclr;          Pointer to function, which clears the interrupt
        UINT imask;         Interrupt mask
}T_DOVR;
```

Overrun handler state reference packet

```
typedef struct t_rovr
{       STAT ovrstat;       Overrun handler state
        OVRTIM leftotm;     Remaining task execution time
}T_ROVR;
```

Version information packet

```
typedef struct t_rver
{       UH maker;           Maker code
        UH prid;            Kernel Identifier code
        UH spver;           ITRON Specification version
        UH prver;           Kernel Version number
        UH prno[4];         Management information
}T_RVER;
```

System state reference packet

```
typedef struct t_rsys
{   INT sysstat;         System state
}T_RSYS;
```

Configuration information packet

```
typedef struct t_rcfg
{   ID tskid_max;        Task ID value upper limit
    ID semid_max;        Semaphore ID value upper limit
    ID flgid_max;        Event flag ID value upper limit
    ID mbxid_max;        Mailbox ID value upper limit
    ID mbfid_max;        Message buffer ID value upper limit
    ID porid_max;        Rendezvous port ID value upper limit
    ID mplid_max;        Variable length memory pool ID value upper limit
    ID mpfid_max;        Fixed length memory pool ID value upper limit
    ID cycno_max;        Cyclic handler ID value upper limit
    ID almno_max;        Alarm handler ID value upper limit
    PRI tpri_max;        Task priority value upper limit
    int tmrqsz;          Task timer queue size
    int cycqsz;          Cyclic handler timer queue size
    int almqsz;          Alarm handler timer queue size
    int istksz;          Interrupt handler stack size (in bytes)
    int tstksz;          Timer event handler stack size (in bytes)
    SIZE sysmsz;         System memory size (in bytes)
    SIZE mplmsz;         memory size of memory-pool (in bytes)
    SIZE stkmsz;         memory size of stack (in bytes)
    ID dtqid_max;        Data queue ID value upper limit
    ID mtxid_max;        Mutex ID value upper limit
    ID isrid_max;        Interrupt service routine (ISR) ID value upper limit
    ID svcfn_max;        upper limit for Extended service call functional number
}T_RCFG;
```

Extended service call definition information

```
typedef struct t_dsvc
{   ATR svcatr;          Extended service call attribute
    FP svcrtn;           Extended service call routine address
    INT parn;            Number of parameters of the extended service call routine
}T_DSVC;
```

7.5 Constant list

Task handler attribute

TA_HLNG	0x0000	Description in high-level language
TA_ACT	0x0002	Task creation in ready state

Task waiting queue attribute

TA_TFIFO	0x0000	FIFO (First-In First-Out)
TA_TPRI	0x0001	Task priority order
TA_TPRIR	0x0004	Receiving task priority order (Message buffer)

Timeout

TMO_POL	0	Polling (without waiting)
TMO_FEVR	-1	Infinite waiting (without timeout)

Task ID

TSK_SELF	0	Specifies task itself
TSK_NONE	0	No task

Task priority

TPRI_INI	0	Priority during initialization
TPRI_SELF	0	Task own base priority
TMIN_TPRI	1	minimum value of the priority
TMAX_TPRI		Maximum priority value (depends on the configuration value)

Task state

TTS_RUN	0x0001	Running state
TTS_RDY	0x0002	Ready state
TTS_WAI	0x0004	WAITING state
TTS_SUS	0x0008	SUSPENDED state
TTS_WAS	0x000c	WAITING-SUSPENDED state
TTS_DMT	0x0010	DORMANT state

Task exception handler state

TTEX_ENA	0x00	Task exception handling allowed
TTEX_DIS	0x01	Task exception handling prohibited

Task wait factor

TTW_SLP	0x0001	Waiting for wakeup
TTW_DLY	0x0002	Fixed time wait
TTW_SEM	0x0004	Waiting for semaphore acquisition
TTW_FLG	0x0008	Waiting for event flag
TTW_SDTQ	0x0010	Waiting for data queue transmission
TTW_RDTQ	0x0020	Waiting for data queue reception
TTW_MBX	0x0040	Waiting for message from mailbox
TTW_MTX	0x0080	Waiting for mutex acquisition
TTW_SMBF	0x0100	Waiting for message buffer message transmission
TTW_MBF	0x0200	Waiting for message buffer message reception
TTW_CAL	0x0400	Waiting for rendezvous call
TTW_ACP	0x0800	Waiting for rendezvous reception
TTW_RDV	0x1000	Waiting for rendezvous end
TTW_MPF	0x2000	Waiting for variable length memory pool acquisition
TTW_MPL	0x4000	Waiting for fixed length memory pool acquisition

Event flag attribute

TA_WSGL	0x0000	multiple task waiting prohibition
TA_CLR	0x0004	Clear flag
TA_WMUL	0x0002	multiple task waiting allowed

Event flag wait mode

TWF_ANDW	0x0000	Waiting with AND logic
TWF_ORW	0x0001	Waiting with OR logic
TWF_CLR	0x0004	Clear flag

Message queue type

TA_MFIFO	0x0000	FIFO (First-In-First-Out) type
TA_MPRI	0x0002	as per message priority

Message priority

TMIN_MPRI	1	Highest priority of message

Mutex attribute

TA_INHERIT	0x0002	Priority inheritance protocol
TA_CEILING	0x0003	Priority maximum limit protocol

Rendezvous port attribute

TA_NULL	0	null

Cyclic handler attribute

TA_STA	0x0002	Cyclic handler start
TA_PHS	0x0004	Phase preservation

Cyclic handler state

TCYC_STP	0x0000	Stop state
TCYC_STA	0x0001	Run state

Alarm handler state

TALM_STP	0x0000	Stop state
TALM_STA	0x0001	Run state

Overrun handler state

TOVR_STP	0x0000	Stop state
TOVR_STA	0x0001	Run state

System state

TSS_TSK	0	Task context part
TSS_DDSP	1	Task context part (Dispatch prohibition state)
TSS_LOC	3	Task context part (CPU lock state)
TSS_INDP	4	Non-task context part

Maximum number of queuing

TMAX_WUPCNT	255	maximum number of wakeup requests by wup_tsk
TMAX_SUSCNT	255	maximum number of task suspend requests by sus_tsk
TMAX_ACTCNT	255	maximum number of wakeup requests by act_tsk
TMAX_MAXSEM	65535	maximum count of Semaphores

Other constants

TRUE	1	Boolean true
FALSE	0	Boolean false

7.6 NORTi3 compatible mode

NORTi Version 4 can be used in NORTi3 compatible mode by defining V3 pre-processor macro. Since norti3.h is included if V3 macro is defined, the system calls of NORTi3 format are usable. After little correction, source file can shift to NORTi3 from NORTi Version 4.

However, as compared to μITRON4.0 specifications, following points are differed.

- The self-task forced-termination (ter_tsk) error code is not E_OBJ but E_ILUSE.
- A self-task wakeup command does not become an error. The request is put into a queue.
- Similar to the event flag, which does not allow the waiting for two or more tasks simultaneously, the error code at the time of simultaneous waiting for two or more tasks by wai_flg is not E_OBJ but E_ILUSE.
- A slef-task suspension command (sus_tsk) will not become an error if it is not in the dispatch prohibition state.
- In case of a mailbox with the queue attribute specification of message priority (not FIFO), the maximum priority becomes the same as that of maximum task priority value.
- The unnecessary information from μITRON4.0 specification (for example extended information), is disregarded. The system call that refers to the object state (ref_xxx), returns back NULL value.
- Since the concept of timeout of tcal_por is changed, pcal_por cannot be used. Moreever, meaning of the timeout in fwd_por is also changed.
- Automatic definition release of alarm handler is not possible.

In addition, please note following points when using NORTi Version 4.

- Automatic ID allocation is assigned sequentially starting from the highest ID number, which can be used.
- In cre_yyy, when ID number 0 is specified, automatic ID allocation is carried out and is not considered as an error.
- Since NORTi3 type object creation information (T_Cxxx type) is changed into NORTi4 type and copied to a system memory, system memory consumption increases.

Index

R

S

T

U

V

W

NORTi Version 4 User's Guide
Kernel Edition

Rev. 1.00	05/Mar/2005	English edition document created
Rev. 1.01	10/Apr/2006	MiSPO company information updated

http://www.mispo.co.jp/
MiSPO Co. Ltd.
Takatsu Park Plaza3F, 5-1-1, Futago, Takatsu-ku, Kawasaki 213-0002, JAPAN
Tel. +81-44-829-3381 fax. +81-44-829-3382
E-mail: sales@mispo.co.jp
Technical support: norti@mispo.co.jp