T-Engine Forum

SMP T-Kernel Specification

Ver. 1.00.00

TEF021-S002-01.00.00/en

February 2009

T-Engine Forum

http://www.t-engine.org/

SMP T-Kernel Specification (Ver.1.00.00)

TEF021-S002-01.00.00/en

February 2009

About this specification, please refer to follows;

Publisher

T-Engine Forum

The 28th Kowa Building 2-20-1 Nishi-gotanda
Shinagawa-Ward Tokyo 141-0031 Japan

TEL：+81-3-5437-0572　FAX：+81-3-5437-2399

E-mail：office@t-engine.org

System Call Notation

In the parts of this specification that describe system calls, the specification of each system call is explained in the format illustrated below.

System call name   Summary description

-System call name:-

[C Language Interface**]**
- Indicates the C language interface for invoking the system call.-

[Parameters**]**
- Describes the system call parameters, i.e., the information passed to the OS when the system call is executed.-

[Return Parameters**]**
- Describes the system call return parameters, i.e., the information returned by the OS when execution of the system call ends.-

[Error Codes**]**
- Describes the errors that can be returned by the system call.-

*The following error codes are common to all system calls and are not included in the error code listings for individual system calls.
E_SYS, E_NOSPT, E_RSFN, E_MACV, E_OACV
*Error code E_CTX is included in the error code listings for individual system calls only when the conditions for its occurrence are clear (e.g., system calls that enter WAIT state). Depending on the implementation, however, the E_CTX error code may be returned by other system calls as well. The implementation-specific occurrences of E_CTX are not included in the error code specifications for individual system calls.

[Description**]**
- Describes the system call functions. -
*When the values to be passed in a parameter are selected from various choices, the following notation is used in the parameter descriptions.

(x ‖ y ‖ z)   - Set one of x, y, or z.
x ｜ y   - Both x and y can be set at the same time (in which case the logical sum of x and y is taken).
s [x]   - x is optional

Example:
When wfmode := (TWF_ANDW ‖ TWF_ORW) ｜ [TWF_CLR], wfmode can be specified in any of the following four ways.

TWF_ANDW
TWF_ORW
(TWF_ANDW | TWF_CLR)
(TWF_ORW | TWF_CLR)

[Additional Notes**]**
- Supplements the description by noting matters that need special attention or caution, etc. -

[Rationale for the Specification**]**
- Explains the reason for adopting a particular specification. –

[Items Concerning SMP T-Kernel]
- Describes sections where the T-Kernel 1.00 Specification differs from SMP T-Kernel.-

Chapter 1 SMP T-Kernel Overview

1.1 Position of SMP T-Kernel

SMP T-Kernel is a real-time operating system for symmetric multiprocessors (SMP: Symmetric Multiple Processor). The functions of SMP T-Kernel have been extended to support SMP on top of the T-Kernel 1.00 Specification for single processor embedded systems.
Multiprocessors have asymmetric multiprocessors (AMP: Asymmetric Multiple Processor) in addition to SMP. The T-Kernel used for AMP is called AMP T-Kernel. SMP T-Kernel and AMP T-Kernel aim to share specifications as much as possible in consideration of compatibility between them. Both are collectively referred to as "MP T-Kernel".

1.2 Background

The necessity of multiprocessors has been increasing along with the increasing size and improving sophistication of embedded systems. In past embedded systems that have used multiprocessors, generally it was not the OS but the application program that handled control and communication between processors in its own scheme. However, in the future it is preferable that this handling should be standardized from the viewpoint of software compatibility and portability. Additionally, high-speed communication between cores has become possible and OS level control between processors is now simpler in recent multicore processors where the cores of multiple processors have been built inside a single chip. Based on the above observation, extension of function of T-Kernel to support multiprocessor systems was examined.

The multiprocessor is classified broadly into AMP and SMP according to the configuration. In AMP, roles are set in each processor statically, and statically assigned programs which include the OS operate in each processor. In SMP, all roles of a processor are equal, and programs are dynamically allocated by the OS to each processor. The functions and implementation of OS are very different in the case of AMP and SMP in this way. Therefore, AMP T-Kernel and SMP T-Kernel were examined separately during the establishment of specifications.

However, as the number of processors increases, it is conceivable an SMP and AMP combination system will appear. Moreover, the demand that would like to see software shared among AMP, SMP, and single processor systems is also significant. Thus, compatibility between AMP T-Kernel and SMP T-Kernel is deemed very important, and the future integration of AMP T-Kernel and SMP T-Kernel is being considered as a result.

1.3 Policies of Specification Establishment

1.3.1 Fundamental Policy

SMP T-Kernel is the real-time OS that mainly targets embedded systems.

In the existing T-Kernel 1.00 Specification, one of the purposes was to improve the portability and distribution of software in various embedded systems.

SMP T-Kernel is a successor of T-Kernel, and improving the portability and distribution of software in various SMP systems is also one of its goals. In addition, the portability and distribution of software with embedded systems that has non-SMP architecture, namely AMP and a single processor are also important.

Based on the above observation, the following fundamental policy was set during the establishment of the SMP T-Kernel Specification.

(1) Compatibility with the Standard T-Kernel

The aim for SMP T-Kernel is to have upper compatibility with the standard T-Kernel at the source code level. The API is to be common with the standard T-Kernel other than the functions extended in SMP T-Kernel, and the porting of software shall be simple. Moreover, the development of software that can run under both SMP T-Kernel and standard T-Kernel shall be made possible.

(2) Reducing the Hardware-Dependency and Supporting Various SMP Systems

The goal for SMP T-Kernel is the support of various types of hardware without depending on the architecture of specific hardware and making porting simple.

(3) Valuing Performance as a Real-time OS Targeting Embedded Systems

The functions of a real-time OS that the T-Kernel 1.00 Specification offers shall be provided by SMP T-Kernel as well.

Therefore, when these functions are used in a single processor system, the goal is execution efficiency equal to that of T-Kernel 1.00 implementation on a single processor. In addition, the communication overhead between processors is paid due attention during design.

1.3.2 Hardware Prerequisites

The hardware prerequisites are stipulated as follows according to the fundamental policy stated above:

(1) Each processor that configures SMP must possess the capability to operate the T-Kernel 1.00 Specification OS on its own. Specifically, CPU shall be 32-bit or more powerful with MMU (Memory Management Unit). Although the MMU is not indispensable, restriction(s) will be imposed on functions without the MMU.

(2) The following functions assumes SMP

Each processor that comprises SMP does not differ in basic function but can execute the same program code and can share the main memory with all other processors. Moreover, when there is a cache function for memory, it has the hardware function to guarantee coherency between processors.

1.3.3 Basic System Configuration

The SMP system consists of multiple processors. All processors are managed by one SMP T-Kernel, and programs to be executed are allocated to each processor dynamically. Task scheduling and object management are uniformly managed in the entire system by SMP T-Kernel.

User programs do not need to be aware of individual processors. In the same manner as T-Kernel user programs operate on single processor systems, programs operate under one SMP T-Kernel.

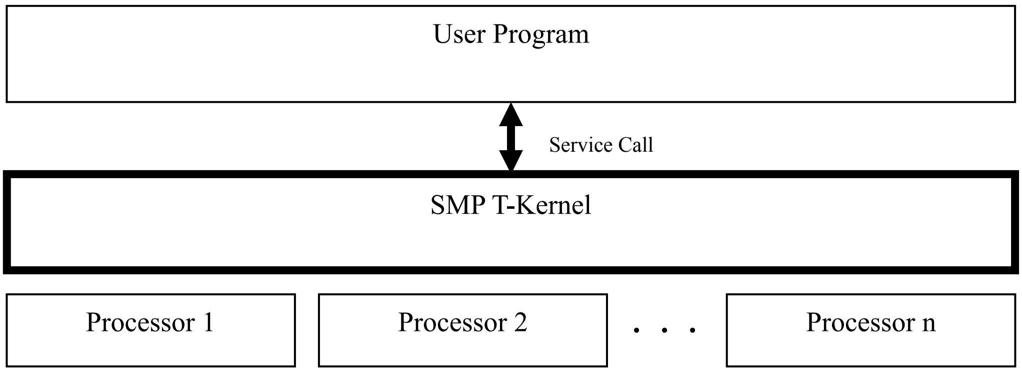


Figure 1: SMP T-Kernel System Configuration Diagram

Like the T-Kernel Specification, individual AMP T-Kernel consists of the following three parts: T-Kernel/OperatingSystem(T-Kernel/OS), T-Kernel/SystemManager(T-Kernel/SM), and T-Kernel/DebuggerSupport (T-Kernel/DS).)

T-Kernel/OperatingSystem (T-Kernel/OS) provides the following functions.

- Task Management functions
- Task-Dependent Synchronization Functions
- Task Exception Handling Functions
- Synchronization and Communication Functions
- Extended Synchronization and Communication Functions
- Memory Pool Management Functions
- Time management functions
- Domain Management Functions
- Interrupt Management Functions
- System Status Management Functions

- Subsystem management functions

T-Kernel/SystemManager (T-Kernel/SM) provides the following functions.

- System memory management functions
- Address space management functions
- Device management functions
- Interrupt management functions
- I/O port access support functions
- Interprocessor Management Functions
- Power management functions
- System configuration information management functions

T-Kernel/DebuggerSupport (T-Kernel/DS) provides the following functions exclusively for the debugger.

- Kernel internal state reference
- Trace

Chapter 2 Concepts Underlying the SMP T-Kernel Specification

2.1 Definition of Basic Terminology

The basic terminology is provided at the inception of the SMP T-Kernel Specification. These terms are common with the T-Kernel 1.00 Specification.

2.1.1 Implementation-Related Terminology

(1) Implementation-defined

There are items that have not been standardized as specifications. Therefore, specifications must be stipulated for each implementation. Specific implementation details must be noted in the implementation specification. In application programs, portability is not assured for sections which are dependent on implementation- defined items.

(2) Implementation-dependent

In the specification, something that is implementation-dependent refers to an item which changes behavior according to the target system or the operating conditions of the system. Behavior must be described for each implementation and specific implementation details must be noted in the implementation specification. In application programs, the sections which depend on implementation-dependent items basically need to be changed when porting.

2.1.2 System-Related Terminology

(1) Device Driver

A device driver is a program that mainly controls hardware.

Each device driver is placed under the management of T-Kernel, and the interface between T-Kernel and the device driver is stipulated in the specifications of T-Kernel. Furthermore, the standard specification of device drivers is stipulated as the T-Kernel Standard Device Driver Specification.

(2) Subsystem

A subsystem is a program that realizes extended system calls (extension SVC), and extends the functions of T-Kernel. The subsystem is placed under the management of T-Kernel, and the interface between T-Kernel and the subsystem is stipulated in the T-Kernel Specification.

(3) T-Monitor

T-Monitor is a program that mainly performs hardware initialization, system startup, exception and interrupt handling, and provision of basic debugging functions.

Initially, T-Monitor starts when the hardware power is turned on (system reset). T-Monitor then initializes the necessary hardware, and starts T-Kernel.

T-Monitor is not part of T-Kernel and is not included in the T-Kernel Specification.

(4) T-Kernel Extension

T-Kernel Extension is a program for extending the functions of T-Kernel and realizes the functions of a more sophisticated OS. T-Kernel Extension has some specifications including T-Kernel Standard Extension as the standard specification.

T-Kernel Standard Extension is implemented as a subsystem of T-Kernel and provides file system and process management functions.

The realization of functions of a more sophisticated OS becomes possible by combining these T-Kernel Extensions with T-Kernel. Moreover, an OS with different functions can be realized by replacing T-Kernel Extension.

(5) Application and System Software

An application is a program created by the user on system software.

System software is a program for operating applications, and it is divided into the hierarchy of T-Monitor, T-Kernel, and T-Kernel Extension from the standpoint of the application. However, T-Monitor and T-Kernel Extension do not always exist. Finally, device drivers are handled as part of T-Kernel.

(6) Kernel Object

A resource which is an operational object of T-Kernel is called a Kernel Object or Object for short. Execution programs such as tasks and synchronization handlers and resources for synchronization and communication such as semaphores and event flags are all Kernel Objects.
The Kernel Object is identified by a numerical ID. For example, the Task ID identifies a task. All Object IDs are dynamically and automatically allocated in T-Kernel during program execution.

2.1.3 Meaning of Other Basic Terminology

(1) Task, invoking task

The basic logical unit of concurrent program execution is called a “task”. While instructions within one task are executed in sequence, instructions within different tasks can be executed in parallel. This concurrent processing is a conceptual view from the standpoint of applications. In reality, multiple executing tasks cannot exceed the number of processors and be truly executed concurrently. In such cases, processing is accomplished by time-sharing among tasks as controlled by the kernel.

A task that invokes a system call is called the “invoking task”.

(2) Dispatch, dispatcher

The switching of tasks executed by the processor is called “dispatching” (or task dispatching). The kernel mechanism by which dispatching is realized is called a “dispatcher” (or task dispatcher).

(3) Scheduling, scheduler

The processing to determine which task to execute next is called “scheduling” (or task scheduling). The kernel mechanism by which scheduling is realized is called a “scheduler” (or task scheduler). Generally a scheduler is implemented inside system call processing or in the dispatcher.

(4) Context

The environment in which a program runs is generally called “context”. For two contexts to be called identical, at the very least, the processor operation mode (Execution mode of the program stipulated by the processor such as privilege and user) must be the same and the stack space must be the same (part of the same contiguous memory area). Note that context is a conceptual view from the standpoint of applications; even when processing must be executed in independent contexts, in actual implementation both contexts may sometimes use the same processor operation mode and the same stack space.

(5) Precedence

The relationship among different execution requests that determines their order of execution is called “precedence”. When a higher-precedence execution request becomes ready for execution while a low-precedence execution request is satisfied and is in progress, as a general rule, the higher-precedence execution request is run ahead of the other request

(6) API and System Calls

The standard interface to call functions provided by T-Kernel from the application and middleware is collectively called API (Application Program Interface). API includes those which are realized as macros and libraries in addition to system calls that call the OS functions directly.

[Additional Note]

Priority is a parameter assigned by an application to control the order of task or message processing. Precedence, on the other hand, is a concept in the specification used to make clear the order in which processing is to be executed. Precedence relation among tasks is determined based on task priority.

2.2 SMP T-Kernel System

2.2.1 Processor

The SMP system is configured with multiple processors. Each processor is identified by Processor ID number. Processor ID numbers are consecutive numbers beginning with 1 and are designated statically as one item of system configuration information during system construction. When a specific processor operates at system startup, the ID number for the processor is 1. The allocation of other numbers is implementation-defined.

Processors are not distinguished for the application by SMP T-Kernel. Therefore, regular applications do not need to be aware of individual processors. Processor ID's are only used to specify a processor in particular when it is absolutely necessary.

2.2.2 Processor and SMP T-Kernel

Processors in SMP T-Kernel are managed by one SMP T-Kernel. For example, the management of kernel objects, task scheduling, system management such as devices and subsystems, and management of resources such as memory are uniformly managed by one SMP T-Kernel.

Tasks are dynamically allocated to each processor by SMP T-Kernel. Applications do not need to be aware of the processor where the program is executed or the number of processors in the system, etc.

However, in the following special cases, programs need to be aware of individual processors.

(1) Control at a level close to hardware such as interrupts and device control, etc.
(2) Execution processor specification of the task

[Additional Notes]

User programs are relieved from multiprocessor control. Processors are not distinguished by SMP T-Kernel.

At the same time user programs do not depend on the number of processors. Programs can have compatibility at the source code level both in SMP T-Kernel systems with different numbers of processors and single processor T-Kernel.

Conversely, hardware-dependency decreases portability. Hence, sufficient attention must be paid to programs that are aware of each processor. Moreover, it is difficult to individually control each processor in SMP systems. If there is a desire to control each processor proactively and individually, the use of AMP T-Kernel must be examined.

2.2.3 Differences With Single Processor Systems

In SMP T-Kernel, the system configuration is equal to single processor T-Kernel in that there are a kernel and an application in the system. The main difference between SMP T-Kernel and single processor T-Kernel is that multiple tasks and handlers may be executed literally at the same time as seen from the application. The following occurs as a result.

(1) Multiple tasks in RUN state exist
The maximum number of running tasks in a single processor T-Kernel is one. However, in SMP T-Kernel, at the maximum, tasks as many as the number of processors can go to RUN state. Therefore, there is a possibility that running tasks are directly and indirectly controlled from other running tasks. This is not possible in single processor systems.
Derived from this, while executing a certain task, a task with lower priority than the executing task may be executed., As a result, the currently executing task may be influenced by this.

(2) Other tasks and handlers may be executed even when a handler is in execution
In single processor T-Kernel, various handlers such as the interrupt handler have higher priority than tasks in execution and tasks are never interrupted by tasks while handlers are in execution. However, in SMP T-Kernel, there is a possibility that tasks and other handlers are executed even during the handler execution.

[Additional Notes]

Single processors can be thought of as a special state with only one processor in SMP. Therefore, programs that do not depend on the number of processors which operate under SMP T-Kernel can also be operated under single processor T-Kernel. In other words, it is possible to write programs that are compatible between single processor T-Kernel and SMP T-Kernel.

However, there is a possibility that existing single processor T-Kernel programs may implicitly conduct mutual exclusion or synchronization control by using the priority of the tasks because they are not aware of multiprocessors. Moreover, embedded systems have the strong tendency of excluding unnecessary controls. Therefore, attention must be paid when porting existing single processor T-Kernel programs to SMP T-Kernel. This is explained further in the following examples.

In single processor T-Kernel applications, during the execution of a certain task, tasks having lower priority are never executed or tasks are never executed during the handler execution. Based on the said principles, implicit mutual exclusion control using priority is often done. In SMP T-Kernel, since implicit mutual exclusion control by precedence doesn't work in principle, system calls must be invoked explicitly to conduct mutual exclusion control.

In single processor T-Kernel applications, tasks are executed one at a time. Based on this, the execution order is forecast from the priority of the tasks and is used for task synchronization. In SMP T-Kernel, explicit synchronization control is necessary because the execution order of tasks changes not only according to the priority of tasks but also according to the number of processors.

An example of the changed execution order of tasks under single processor T-Kernel and under SMP T-Kernel is provided below. Here, Task B and Task C are started by Task A. The task priority is Task A > Task B > Task C.

In single processor T-Kernel, tasks are sequentially executed one at a time according to priority [Figure 2(a)].

In SMP T-Kernel, there is the possibility that Task B begins execution immediately when it is started from Task A without waiting for the completion of Task A. For example, if there are three or more processors, and there are no tasks having higher priority than Task A, B, and C, Task B and Task C are executed immediately when they are started. [Figure 2(b)].

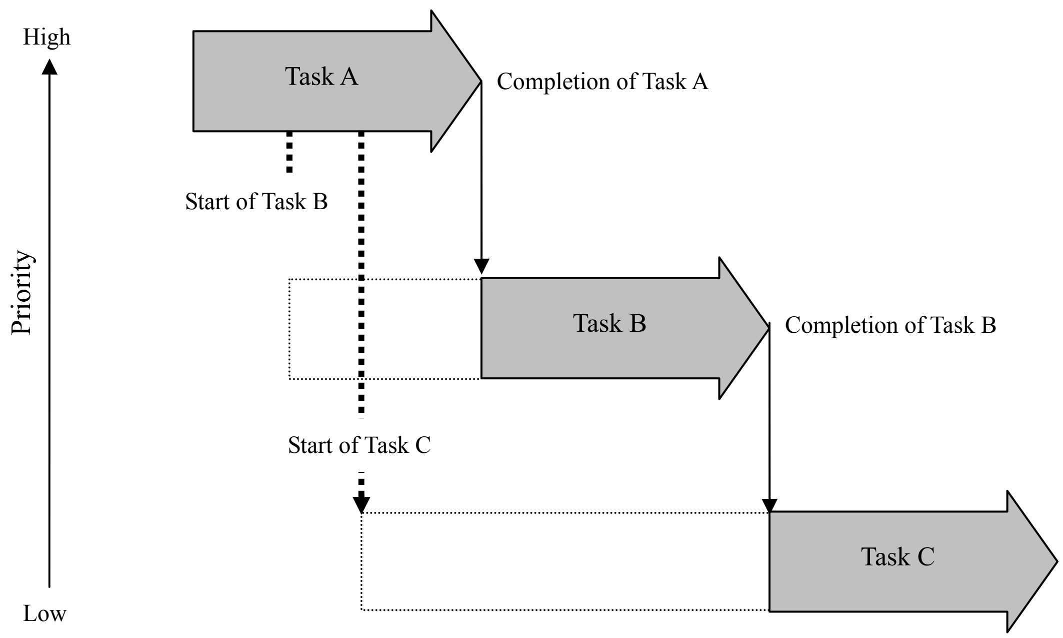


Figure 2(a) Example of task execution by a single processor T-Kernel

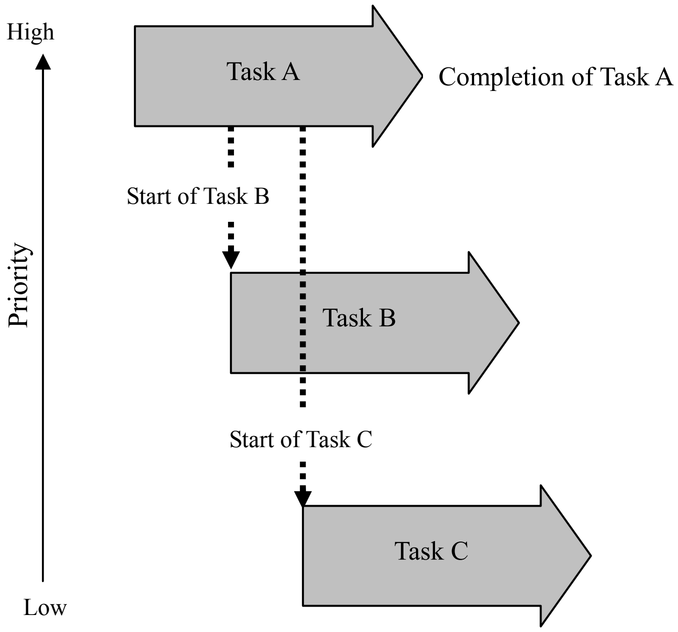


Figure 2(b) Example of task execution by SMP T-Kernel

Here, suppose processing of Task A must end when Task B starts running, and also suppose the processing of Task A and Task B must end when Task C starts running. Therefore, assuming it is a single processor T-Kernel, even if special synchronization is not controlled, processing is executed as expected in this example. However, in SMP T-Kernel, special synchronization control must be conducted using system calls.

Programs that explicitly performs mutual exclusion control and synchronization control can be operated independent of the number of processors, are on single processor T-Kernel, for example. If portability is considered, it is better to explicitly conduct mutual exclusion control and synchronization control.

2.3 Task States and Scheduling Rules

2.3.1 Task States

States of individual tasks of SMP T-Kernel are from the same as those of the T-Kernel 1.00 Specification. However, it must be noted that the number of tasks in RUN state can be up to the number of processors in SMP T-Kernel while only one task is the RUN state in the T-Kernel 1.00 Specification that operates on a single processor.

Task states are classified primarily into the five below. Of these, the Wait state in the broad sense is further classified into three states. Saying that a task is in a Run state means it is in either RUN state or READY state.

(a) RUN state

The task is currently being executed. When a task-independent portion is executing, except when otherwise specified, the task that was executing prior to the start of task-independent portion execution is said to be in RUN state.

(b) READY state

The task has completed preparations for running, but cannot run because a task with higher precedence is running. In this state, the task is able to run whenever it becomes the task with a higher precedence than tasks currently running.

(c) Wait states

The task cannot run because conditions for running are not in place. In other words, the task is waiting for the conditions for its execution to be met. While a task is in one of the Wait states, the program counter, register values, and other information representing the program execution state are saved. When the task resumes running from this state, the program counter, registers and other values revert to their values immediately prior to going into the Wait state. This state is subdivided into the following three states.

(c.1) WAIT state

Execution is stopped because a system call was invoked that interrupts execution of the invoking task until some condition is met.

(c.2) SUSPEND state

Execution was forcibly interrupted by another task.

(c.3) WAIT-SUSPEND state

The task is both in WAIT state and SUSPEND state at the same time. WAIT-SUSPEND state results when another task requests suspension of a task already in WAIT state. T-Kernel makes a clear distinction between WAIT state and SUSPEND state. A task cannot go to SUSPEND state on its own.

(d) DORMANT state

The task has not yet been started or has completed execution. While a task is in DORMANT state, information representing its execution state is not saved. When a task is started from DORMANT state, execution starts from the task start address. Except when otherwise specified, the register values are not saved.

(e) NON-EXISTENT state

A virtual state before a task is created, or after it is deleted, and is not registered in the system.

Depending on the implementation, there may also be transient states that do not fall into any of the above categories (see section 2.4).

When a task going to READY state has higher precedence than the currently running task, a dispatch may occur at the same time as the task goes to READY state and it may make an immediate transition to RUN state. In such a case, the task that was in RUN state up to that time is said to have been preempted by the task newly going to RUN state. Note also that in explanations of system call functions, even when a task is said to go to READY state, depending on the task precedence it may go immediately to RUN state further.

Task starting means transferring a task from DORMANT state to READY state. A task is therefore said to be in a “started” state if it is in any state other than DORMANT or NON-EXISTENT. Task exit means that a task in a started state goes to DORMANT state.

Task wait release means that a task in WAIT state goes to READY state, or a task in WAIT-SUSPEND state goes to SUSPEND state. The resumption of a suspended task means that a task in SUSPEND state goes to READY state, or a task in

WAIT-SUSPEND state goes to WAIT state.

Task state transitions in a typical implementation are shown in Figure 3. Depending on the implementation, there may be other states besides those shown here.

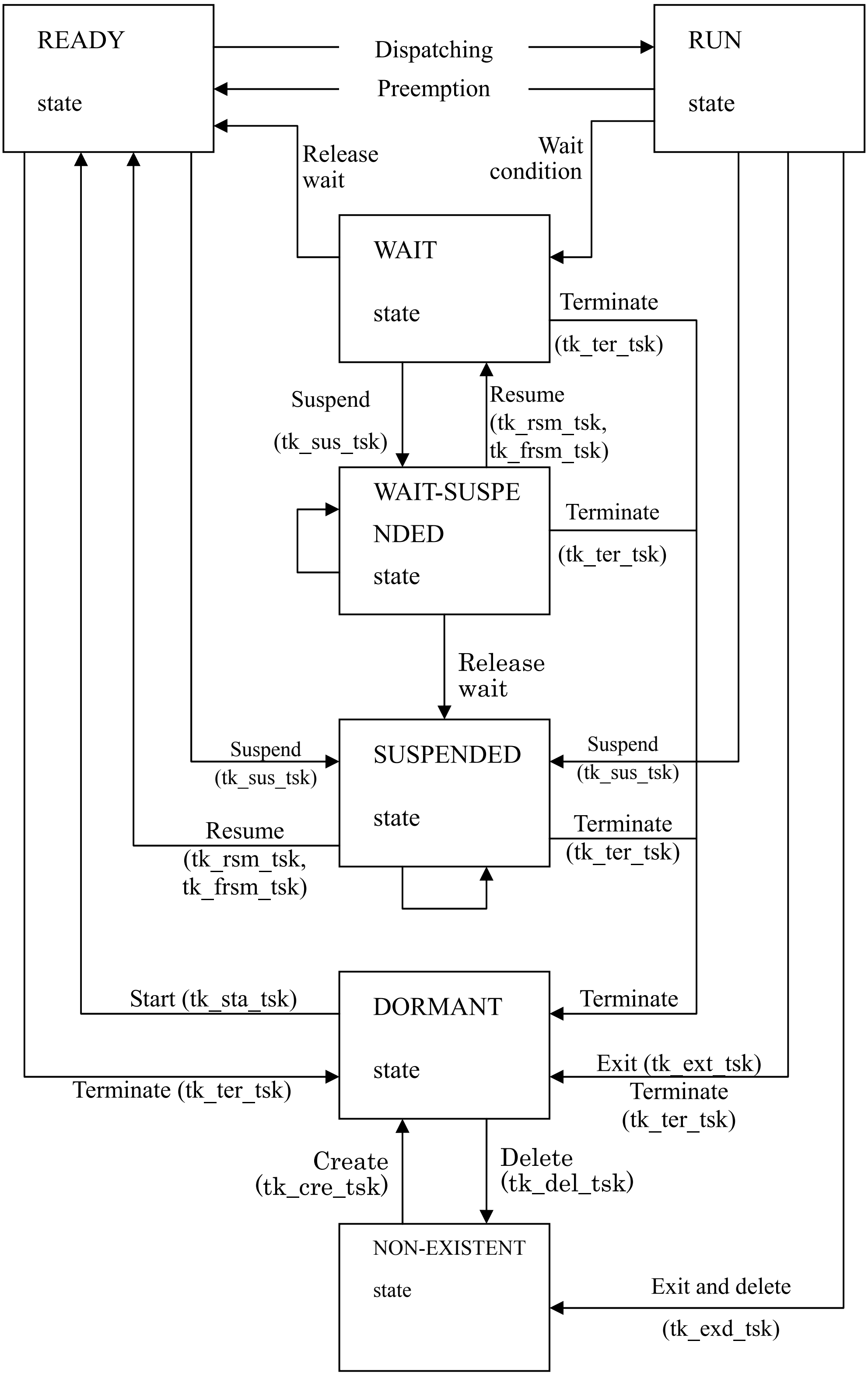


Figure 3: Task State Transitions

A feature of T-Kernel is the clear distinction made between system calls that perform operations affecting the invoking task and those whose operations affect other tasks (see Table 1). The reason for this is to clarify task state transitions and facilitate understanding of system calls.

Table 1 State Transitions Distinguishing Invoking Task and Other Tasks

	Operations in invoking task	Operations on other tasks
Task transition to a wait state (including SUSPEND)	tk_slp_tsk RUN ↓ WAIT	tk_sus_tsk RUN, READY, WAIT ↓ ↓ SUSPEND, WAIT-SUSPEND
Task exit	tk_ext_tsk RUN ↓ DORMANT	tk_ter_tsk RUN, READY, WAIT ↓ DORMANT
Task deletion	tk_exd_tsk RUN ↓ NON-EXISTENT	tk_del_tsk DORMANT ↓ NON-EXISTENT

[Additional Notes]

WAIT state and SUSPEND state are orthogonally related in that a request for transition to SUSPEND state cannot have any effect on the conditions for task wait release. That is, the task wait release conditions are the same whether the task is in WAIT state or WAIT-SUSPEND state. Thus even if transition to SUSPEND state is requested for a task that is in a state of waiting to acquire some resource (semaphore resource, memory block, etc.), and the task goes to WAIT-SUSPEND state, the conditions for allocation of the resource do not change but remain the same as before the request to go to SUSPEND state.

[Rationale for the Specification]

The reason the T-Kernel specification makes a distinction between WAIT state (wait caused by the invoking task) and SUSPEND state (wait caused by another task) is that these states sometimes overlap. By distinguishing this overlapped state as WAIT-SUSPEND state, the task state transitions become clearer and system calls are easier to understand. On the other hand, since a task in WAIT state cannot invoke a system call, different types of WAIT state (e.g., waiting for wakeup, or waiting to acquire a semaphore resource) will never overlap. Since there is only one kind of wait state caused by another task (SUSPEND state), the T-Kernel specification treats overlapping of SUSPEND states as nesting, thereby achieving clarity of task state transitions.

2.3.2 Task Scheduling Rules

When the priority level of a task is changed due to a system call, etc. in T-Kernel, task scheduling is performed. A dispatch occurs when a task in RUN state changes its state due to scheduling. Task scheduling is a preemptive priority-based scheduling based on priority levels assigned to each task. Task scheduling between tasks having the same priority is done on a FCFS (First Come First Served) basis.

The task scheduling of SMP T-Kernel uses a similar method to single processor T-Kernel. However, in SMP T-Kernel, it is different from single processor T-Kernel in that multiple tasks can be in RUN state at the same time.

The following paragraphs will first explain task scheduling in single processor T-Kernel and then will explain task scheduling in SMP T-Kernel.

(1) Task scheduling in single processor T-Kernel

Task scheduling in single processor T-Kernel is equal to task scheduling in the special case of SMP T-Kernel with only one processor. Task scheduling is conducted as follows:

- Precedence is given to tasks that can be executed. Among tasks having different priorities, a task having higher priority has higher precedence. Among tasks having the same priority, the one first going to a run state (RUN state or READY state) has the highest precedence. It is possible, however, to use a system call to change the precedence relation among tasks having the same priority.
- The task with the highest precedence goes to RUN state, and other tasks goes to READY state.
- When the task with the highest precedence changes from one task to another, a dispatch occurs immediately and the task in RUN state is switched. If dispatching is not allowed, however, the switching of the task in RUN state is held off until dispatch occurs.

In other words, tasks that can be executed are considered to be in a queue according to precedence. If the change in the precedence relation among tasks is allowed and the first task in the queue is thus replaced, dispatch occurs.

The task scheduling in single processor T-Kernel is described using the example in Figure 4.

Figure 4(a) illustrates the precedence relation among tasks after Task A of priority 1, Task E of priority 3, and Tasks B, C and D of priority 2 are started in that order. The task with the highest precedence, Task A, goes to RUN state.

When Task A ends, Task B with the next-highest precedence goes to RUN state (Figure 4(b)). When Task A is again started, Task B is preempted and reverts to READY state; but since Task B went to a run state earlier than Task C and Task D, it still has the highest precedence among tasks having the same priority. In other words, the task precedence reverts to that in Figure 4(a).

Next, consider what happens when Task B goes to WAIT state in the conditions in Figure 4(b). Since task precedence is defined among tasks that can be run, the precedences of tasks become as shown in Figure 4(c). Thereafter, when the Task B’s wait state is released, Task B goes to RUN state after Task C and Task D, and thus will have the lowest precedence among tasks of the same priority (Figure 4(d)).

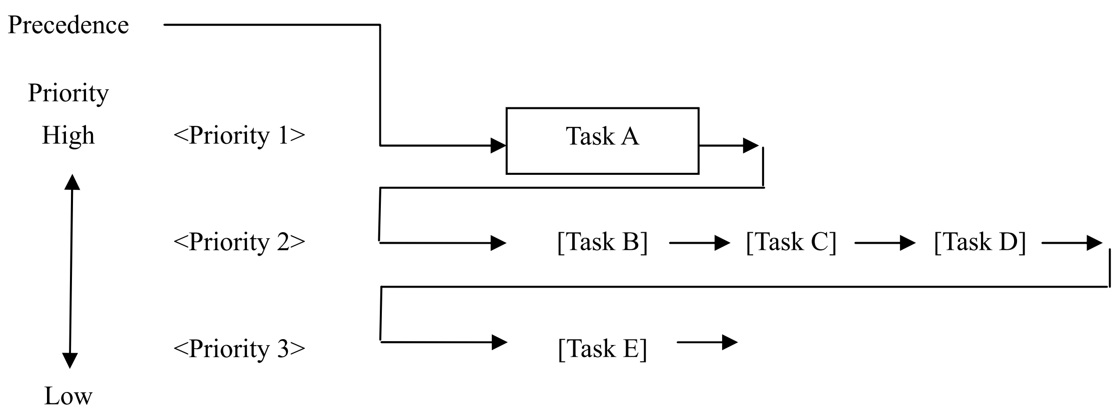


Figure 4(a): Precedence in Initial State

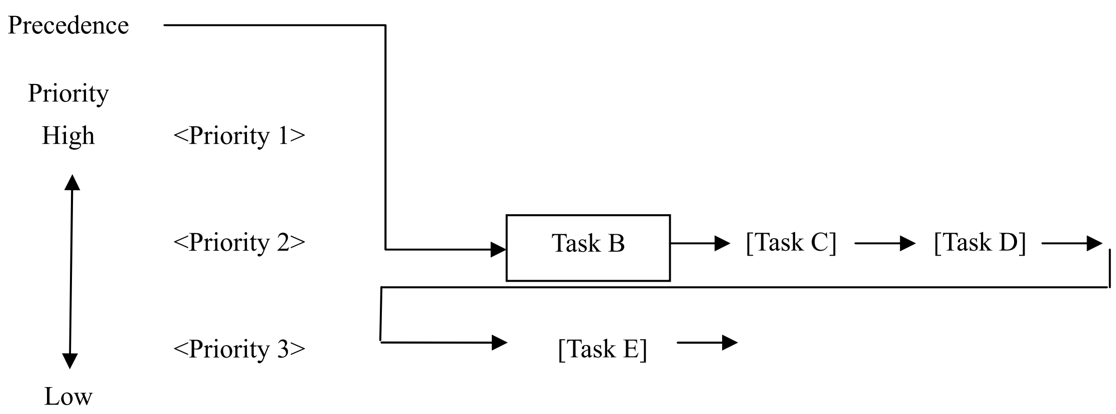


Figure 4(b): Precedence After Task B Goes To RUN State

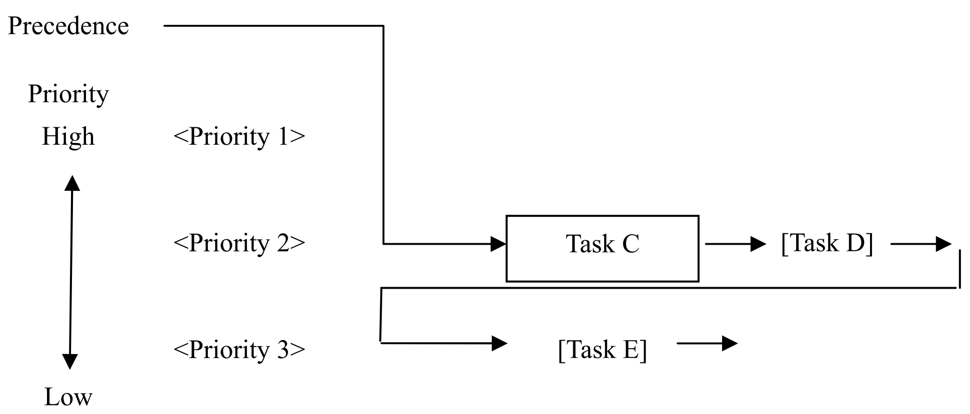


Figure 4(c): Precedence After Task B Goes To WAIT State

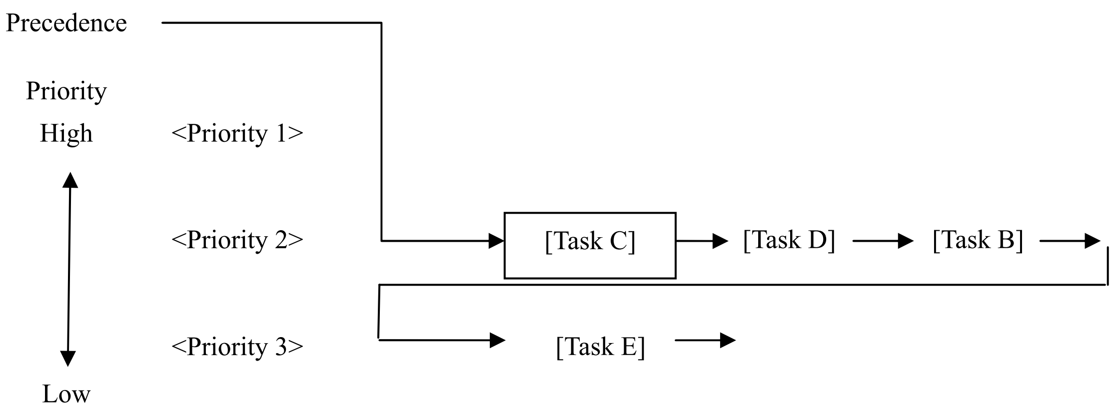


Figure 4(d): Precedence After Task B WAIT State Is Released

Summarizing the above, immediately after a task that went from READY state to RUN state reverts to READY state, it has the highest precedence among tasks of the same priority; but after a task went from RUN state to WAIT state and then the wait is released, its precedence is the lowest among tasks of the same priority.

(2) Task scheduling in SMP T-Kernel

The difference between task scheduling of SMP T-Kernel and of single processor T-Kernel is that the task having the highest precedence goes to RUN state in single processor T-Kernel while, in SMP T-Kernel, tasks equal to the number of execution processors go to RUN state in the order of precedence. The number of processors which can execute tasks is as many as the number of processors that comprise SMP.

Task scheduling is conducted as follows:

- Precedence is given to tasks that can be executed. The rules regarding precedence are the same as those in single processor T-Kernel.
- In the order of precedence, tasks as many as the number of processors that can execute go to RUN state, and other tasks go to READY state.
- When the precedence changes and a task with higher precedence than any of the tasks currently in RUN state appears, a dispatch occurs immediately, and the tasks in RUN state is switched. However, when the tasks in RUN state is in a state where dispatch is not allowed, the switching of the tasks in RUN state is held off until dispatch is allowed.

The task scheduling in SMP T-Kernel is described using the example in Figure 5. Assume that there are two processors. Here, the example of the single processor T-Kernel used in Figure 4 is treated under SMP T-Kernel with two processors.

Figure 5(a) illustrates the precedences of tasks after Task A of priority 1, Task E of priority 3, and Tasks B, C and D of priority 2 are started in that order. In this state, two task, i.e. Task A and Task B, are put in RUN state in the order of precedence (Two is the number of processors).

When Task A exits, Task B and Task C with the next highest precedence will be in RUN state. Task B continues in RUN state and Task C goes to RUN state as shown in Figure 5(b). Thereafter when Task A is started again, Task C is preempted and reverts to READY state; but at this time, there is no change in precedences of Task B, Task C, and Task D. That is, the precedence relation among tasks reverts to as shown in Figure 5(a).

Next, consider what happens when Task B goes to WAIT state in the conditions in Figure 5(b). Since task precedence is defined among tasks that can be run, the precedence relation among tasks becomes as shown in Figure 5(c). Thereafter, when the Task B’s wait state is released, Task B goes to run state after Task C and Task D, and thus will have the lowest precedence among tasks of the same priority (Figure 5(d)).

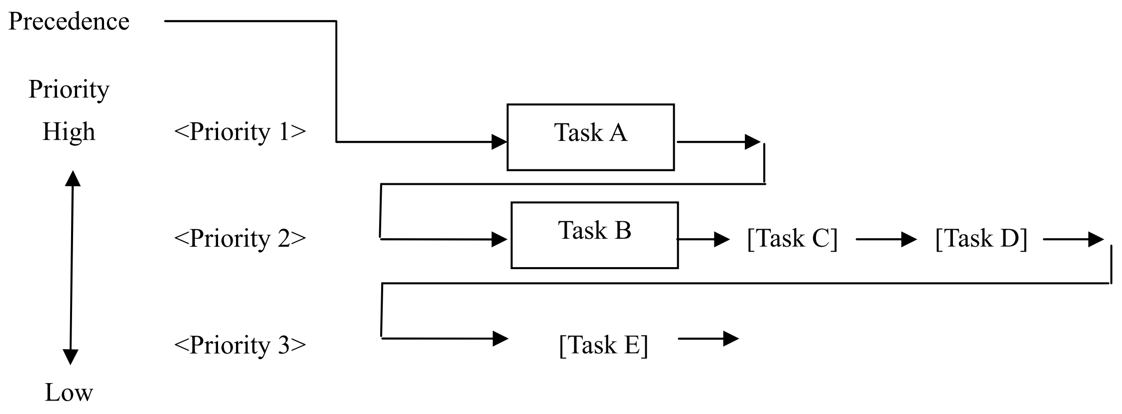


Figure 5(a) Precedence in Initial State

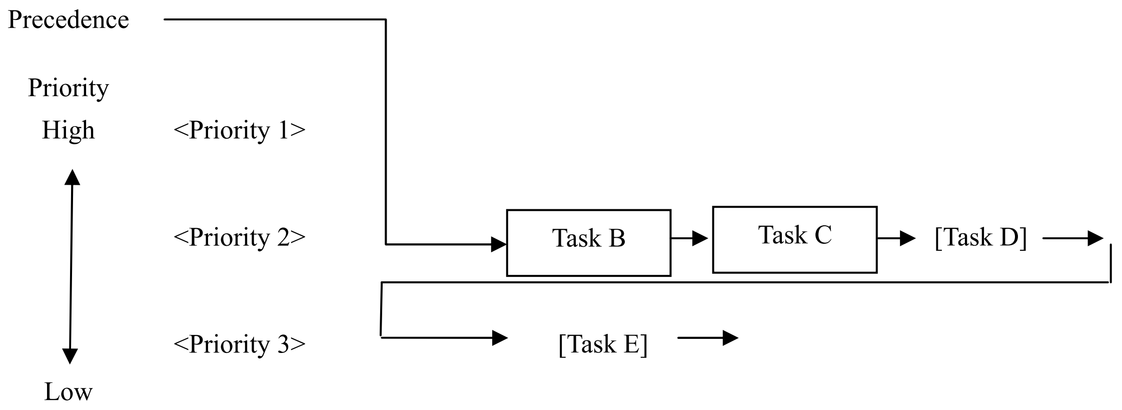


Figure 5(b) Precedence After Task A Ends

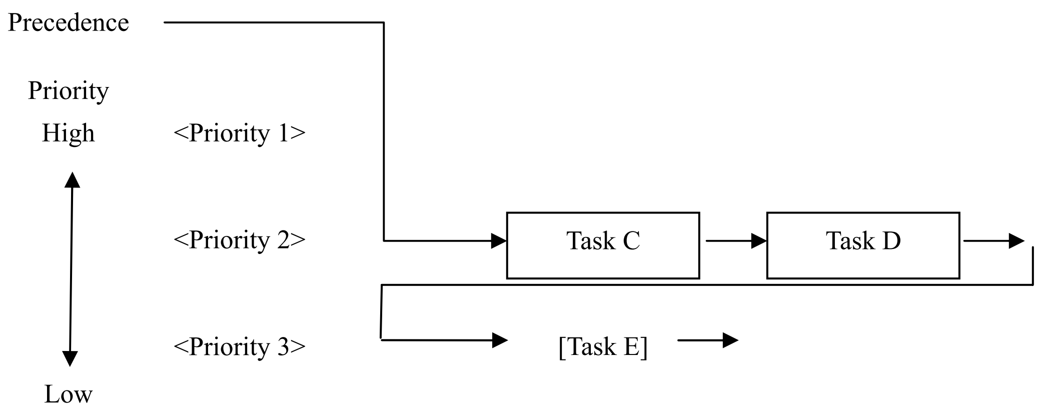


Figure 5(c) Precedence After Task B Goes To WAIT State

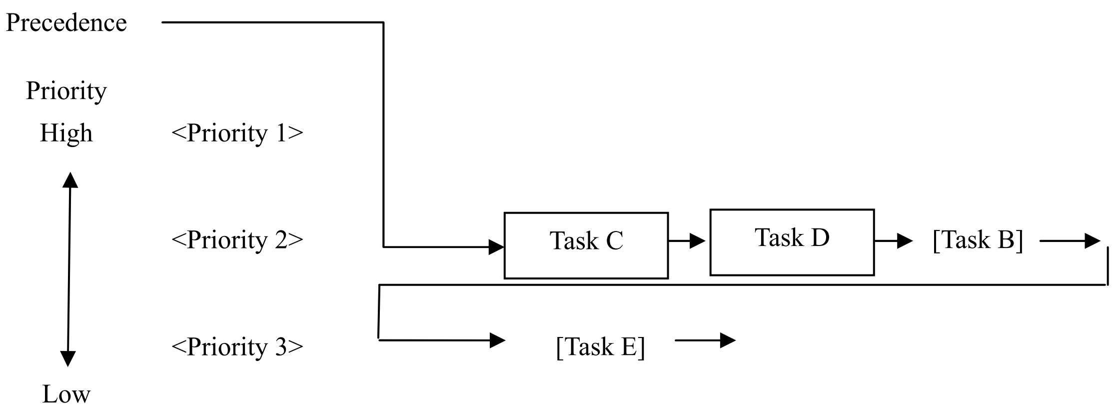


Figure 5(d) Precedence After Task B WAIT State Is Released

In SMP T-Kernel, the dispatch of multiple tasks may occur in a single scheduling step. When this happens, the dispatch of each task is synchronized. For example, when the states of multiple tasks are changed by a single system call, all the task state transitions finish at the time of this system call returns. However, for processors executing the task-independent portion such as interrupt handlers, dispatch is delayed because the dispatch of tasks cannot be executed immediately until the task-independent portion ends.

Tasks which have not been dispatched, in other words, which are continuously in RUN state in the same processor, are not affected by the dispatch of other tasks.

Tasks in dispatch-disabled state are excluded from scheduling. Therefore, a task in dispatch-disabled state always continues in the RUN state on the same processor even after scheduling,

[Additional Notes]

According to the scheduling rules adopted in the single processor T-Kernel specification, so long as there is a high precedence task in a run state, a task with lower precedence will simply not run. That is, unless the highest-precedence task goes to WAIT state or cannot run for other reason, other tasks are not run. This is a fundamental difference from TSS (Time Sharing System) scheduling in which multiple tasks are treated in a fair and equal manner.

In the same way, in SMP T-Kernel, tasks with low precedence in READY state are not executed unless a task with a higher precedence cannot run any more due to reasons such as going to WAIT state.

It is possible, however, to issue a system call for changing the precedence relation among tasks having the same priority. An application can use such a system call to realize round-robin scheduling which is a typical scheduling method used in TSS.

2.3.3 Task Execution Processor

In SMP T-Kernel, tasks as many as the number of execution processors can go to RUN state in the order of precedence of tasks.

The processor to which a task is allocated is implementation-defined and applications do not need to be aware of this information. However, users can specify the execution processor of a task.

In task scheduling, it is actually guaranteed that tasks which continue in RUN state will continue to be allocated to the same processor. However, when scheduling includes tasks which specify the execution processor, this guarantee ends. In other words, a switching of the allocation of the tasks that continue in RUN state to other processors can occur.

The allocation of a task to a processor is explained using Figure 6 as an example of a case where the execution processor is not specified. Here, the number of processors is four and Task A with priority 1, Task B with priority 2, Task C with priority 3 and Task D with priority 4 are in RUN state. All tasks do not have an execution processor specified, and there are no other tasks that can run (Figure 6(a)).

Task E with priority 2 is started then. Task A, Task B, Task C, and Task E go to RUN state and Task D goes to READY state because of the order of their precedence (Figure 6(b)). In other words, a dispatch occurs and Task D and Task E are switched on processor 4.

Prior to and following this scheduling, Task A, Task B, and Task C continue in RUN state. These tasks continue to be allocated to the same processor without dispatch (Figure 6(c)).

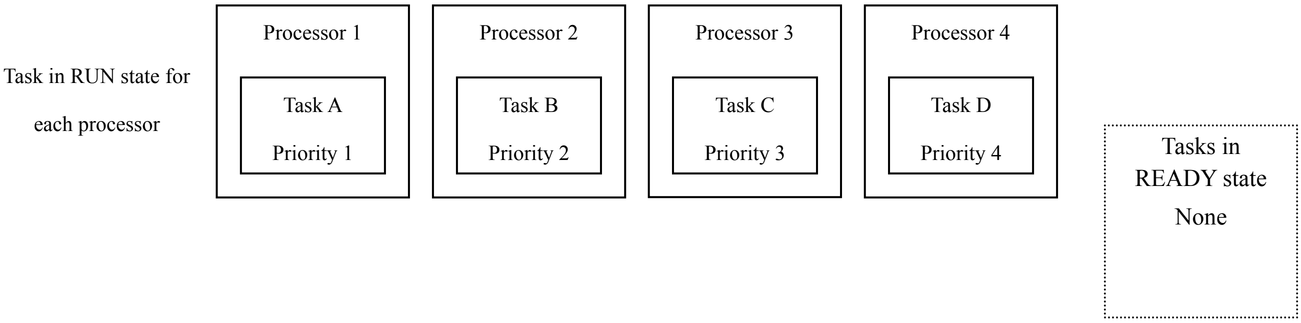


Figure 6(a) Allocation of execution processors　Initial state

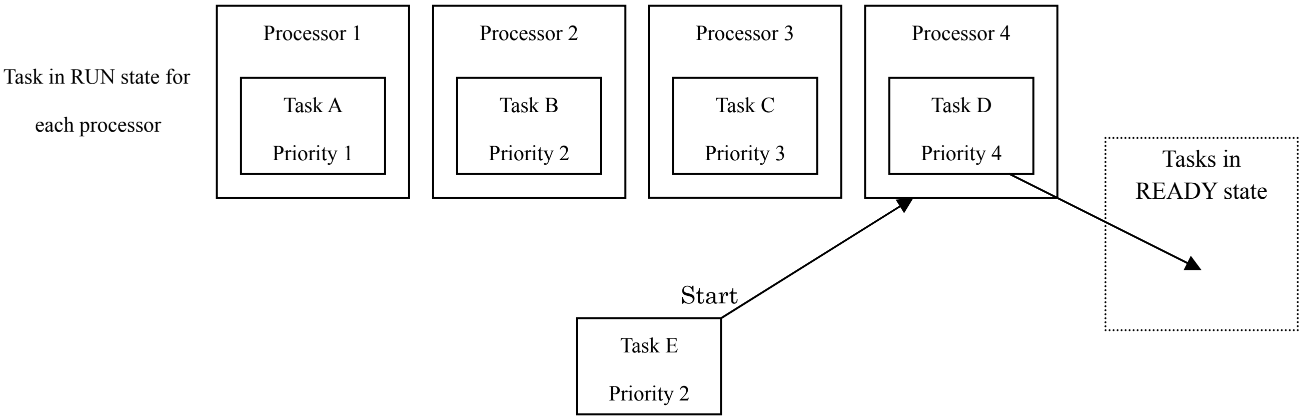


Figure 6(b) Allocation of execution processors　Task E starts and is initialized

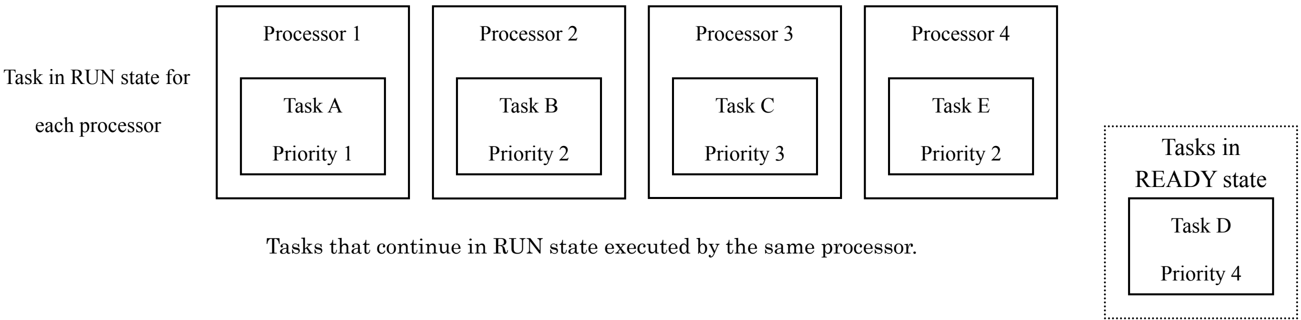


Figure 6(c) Allocation of execution processors　State following a dispatch

Next is an example for the case in which the execution processor of a task is specified.

In SMP T-Kernel, normally tasks are automatically allocated to a processor but processors for executing a task, namely execution processors can be specified by the user. When a task is created, one or more execution processors can be specified. Tasks with a specified execution processor are only executed in the specified processor.

When an execution processor is specified, task restrictions result in the following scheduling.

- There is a possibility that the precedence relation of the RUN and READY tasks is not observed in a normal manner during scheduling and is used in an inverted manner.

When multiple tasks specify the same processor for the execution processor, they may not go to RUN state even though the precedence is high.

Figure 7, as an example, outlines a precedence inversion accompanying the execution processor specifications. Assuming the five tasks of Task A, B, C, D, and E exist in this order of precedence. If the number of processors is four, and execution processors are not specified for the tasks, the four tasks of Task A, B, C, and D go to RUN state. In this case, the precedence relation of the tasks is observed normally [Figure 7(a)].

However, assuming Task A and Task B have specified processor 1 as the execution processor. In this case, Task B does not go to RUN state because Task A is allocated to processor 1 first. Therefore, Task C, D, and E which have lower precedence than Task B go to RUN state, Task B remains in READY state, and an inversion of precedence occurs [Figure 7(b)].

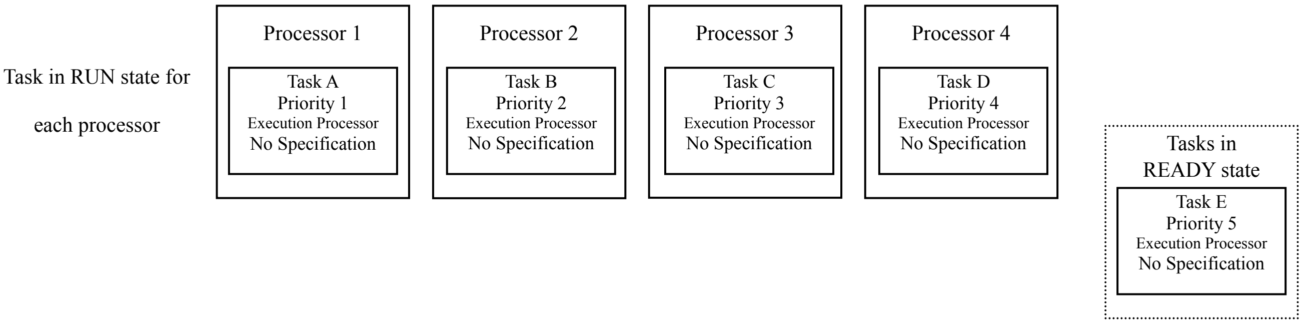


Figure 7(a) Processor allocation when the execution task is not specified

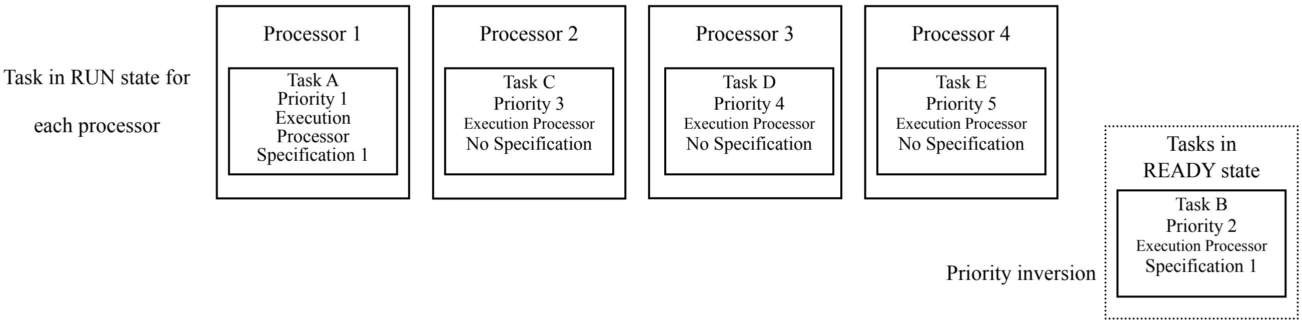


Figure 7(b) Processor allocation when the execution task is specified

- There is a possibility that the processor is switched even when the task continues in RUN state

In task scheduling, if there are tasks with execution processor specification among tasks which go to RUN state, the execution processor can be reallocated and dispatch may occur even if the tasks should continue in RUN state prior to and following the scheduling.

Figure 8, as an example, outlines a processor switching during scheduling accompanying the execution processor specification. Assuming there are four processors and Task A, B, C, and D are in RUN state. Priority levels of tasks are as follows; Task A's priority is 1, Task B's is 2, Task C's is 3, and Task D's is 4. None of the tasks has an execution processor specified.

Here, when Task E with priority 2 is started, dispatch occurs, Task E goes to RUN state, and Task D goes to READY state. If an execution processor is not specified for Task E, Task A, B, and C are not affected and only Task E and D are switched on processor 4. However, when the execution processor is specified for Task E, and if the execution processor is different from the processor executing Task D, the processor is re-allocated and there is a possibility that the processor executing Task A, B, and C is also switched.

Moreover, if the processor of a task in RUN state is changed, and when dispatch is delayed during the execution of the task-independent portion on the original processor, the dispatch will also be delayed in the processor to which the task will be newly allocated.

Correspondence of the tasks and processors when reallocation occurs due to the change of the processor of a task are implementation-defined.

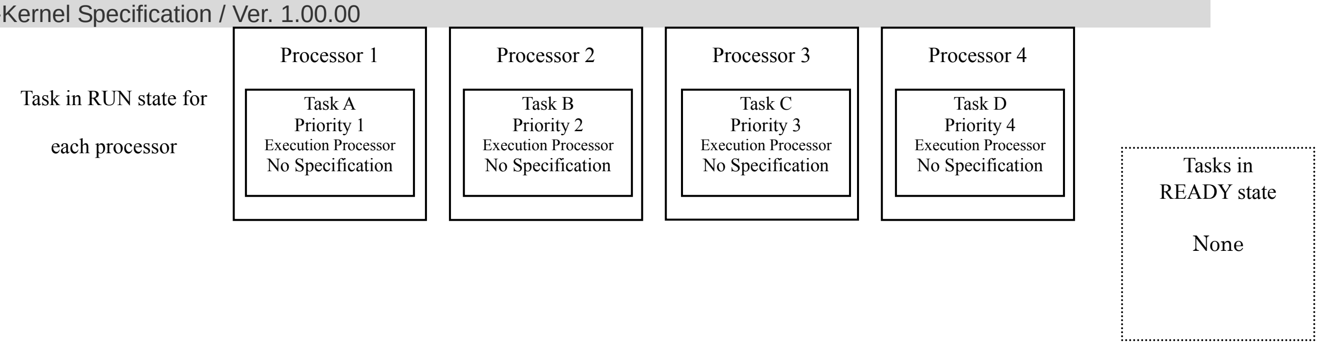


Figure 8(a) Execution processor reallocation　Initial state

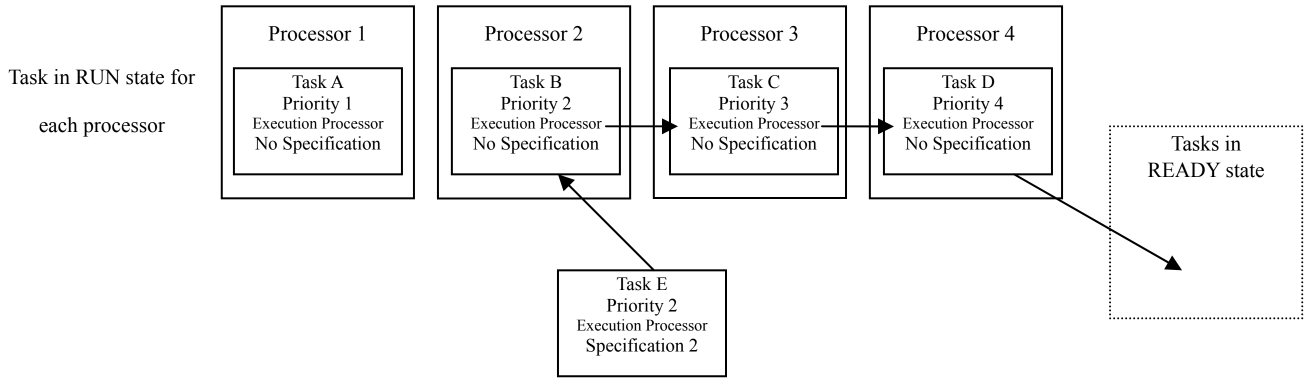


Figure 8(b) Execution processor reallocation　Task E starts and is initialized

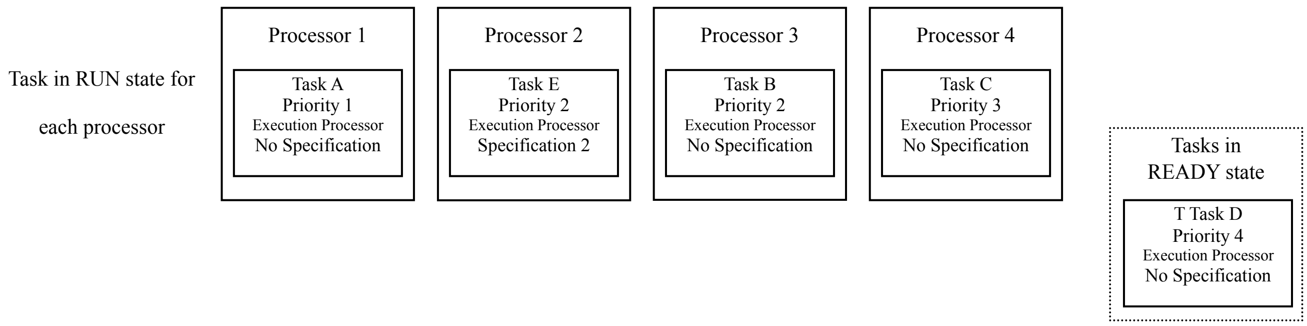


Figure 8(c) Execution processor reallocation　State following dispatch

2.4 System States

There is no difference in the system state of SMP T-Kernel with the T-Kernel 1.00 Specification.
However, the system state in SMP T-Kernel exists in each program executed by individual processors comprising a system.

2.4.1 System States While Non-task Portion Is Executing

When programming tasks to run on T-Kernel, the changes in task states can be tracked by looking at a task state transition diagram. In the case of routines such as interrupt handlers or extended SVC handlers, however, the user must perform programming at a level closer to the kernel than ordinary tasks. In this case, consideration must be made also of system states while a non-task portion is executing, otherwise programming cannot be done properly. An explanation of T-Kernel system states is therefore given here.

System states are classified as in Figure 9.

Of these, a “transient state” is equivalent to OS running state (system call execution). From the standpoint of the user, it is important to note that each of the system calls issued by the user be executed indivisibly, and that the internal states while a system call is executing cannot be seen by the user. For this reason, the state while the OS is running is considered a “transient state” and it is treated as a black box.

In the following cases, however, a transient state is not executed indivisibly.

- When memory is being allocated or freed in the case of a system call that gets or releases memory (while a T-Kernel/SM system memory management function is called).
- In a virtual memory system, when nonresident memory is accessed during system call processing.

When a task is in a transient state such as these, the behavior of a task termination (tk_ter_tsk) system call is not well-defined. Moreover, task suspension (tk_sus_tsk) may cause a deadlock or other problems by stopping the task without clearing the transient state.

Accordingly, as a rule, tk_ter_tsk and tk_sus_tsk should not be used in programs. These system calls should be used only in a subsystem such as a debugger that is so close to OS layer that it can be thought of as part of the OS.

A task-independent portion and quasi-task portion are states while a handler is executing. The part of a handler that runs in a task context is a quasi-task portion, and the part with a context independent of a task is a task-independent portion. An extended SVC handler, which processes extended system calls defined by the user, is executed as a quasi-task portion when it is called from tasks. An interrupt handler or time event handler triggered by an external interrupt is executed as a task-independent portion. In a quasi-task portion, tasks have the same kinds of state transitions as ordinary tasks and system calls can be issued that transition tasks even to WAIT state.

A transient state, task-independent portion and quasi-task portion are together called a non-task portion. When ordinary task programs are running, outside of these, this is “task portion running” state.

- System state
  - Non-task portion running
    - Transient state
      OS running, etc.
    - Task-independent portion running
      Interrupt handler, etc
    - Quasi-task portion running
      Extended SVC handler (OS extended part), etc.
  - Task portion running

Figure 9: Classification of System States

2.4.2 Task-Independent Portion and Quasi-Task Portion

A feature of a task-independent portion (interrupt handlers, time event handlers, etc.) is that because the task that was running immediately prior to entering a task-independent portion is unrelated and independent, the concept of “invoking task” does not exist. Accordingly, a system call that enters WAIT state, or the ones that implicitly specify the invoking task, cannot be called from a task-independent portion. Moreover, the task-independent portion is executed in preference to all tasks. Therefore, execution of task-independent part is not interrupted by dispatching (task switching). If dispatching is necessary, it is delayed until processing leaves the task-independent portion. This is called delayed dispatching. However, dispatching is immediately performed on a processor that is not executing the task-independent portion.

A feature of the quasi-task portion is that it is invoked from the task executing prior to entering the quasi-task portion (the requesting task), and the task state of the requesting task continues; moreover, it is even possible to go to WAIT state while in a quasi-task portion. Accordingly, dispatching occurs in a quasi-task portion in the same way as in ordinary task. As a result, even though the OS extended part and other quasi-task portion are a non-task portion, its execution does not necessarily have higher priority at all times over the task portion. This is in contrast to interrupt handlers, which must always be given execution precedence over ordinary tasks.

The example in Figure 10 describes the difference between a task-independent portion and quasi-task portion. In this example, the number of processors is one, and the tasks are Task A (low priority) and Task B (high priority) only.

- An interrupt occurs while Task A (low priority) is running, and in its interrupt handler X (task-independent portion), tk_wup_tsk is issued for Task B (high priority). In accordance with the principle of delayed dispatching, however, dispatching does not yet occur at this point. Instead, after tk_wup_tsk execution, first the remaining part of the interrupt handler is executed. Only when tk_ret_int is executed at the end of the interrupt handler X, dispatching occurs, causing Task B to run [Figure 10(a)].
- In the previous example, assume further that multiple interrupts are permitted, and interrupt handler Y is invoked while interrupt handler X is executing. The principle of delayed dispatching is applied, and there is no dispatching to the return of tk_ret_int from interrupt handler Y either. Dispatching occurs for the first time after the return of tk_ret_int from interrupt handler X and task B is executed [Figure 6(b)].
- Assume that an extended system call is executed in Task A (low priority), and in its extended SVC handler (quasi-task portion), tk_wup_tsk is issued for Task B (high priority). In this case, the principle of delayed dispatching is not applied, so dispatching occurs in tk_wup_tsk processing. Task A goes to READY state in a quasi-task portion, and Task B goes to RUN state. Task B is therefore executed before the rest of the extended SVC handler is completed. The rest of the extended SVC handler is executed after dispatching occurs again and Task A goes to RUN state. [Figure 10(c)].

In reality, tasks can be executed on multiple processors in SMP T-Kernel. Therefore, in the previous example, if tasks woken up by the interrupt handler with tk_wup_tsk can be executed on other processors, dispatching is not delayed. Dispatching occurs on other processors immediately (Thus, delayed dispatching is not done.).

Moreover, the delayed dispatching during execution of the task-independent portion may be affected by scheduling done on other processors. Delayed dispatching is based on the precedence of the task when the execution of task-independent portion is finished.

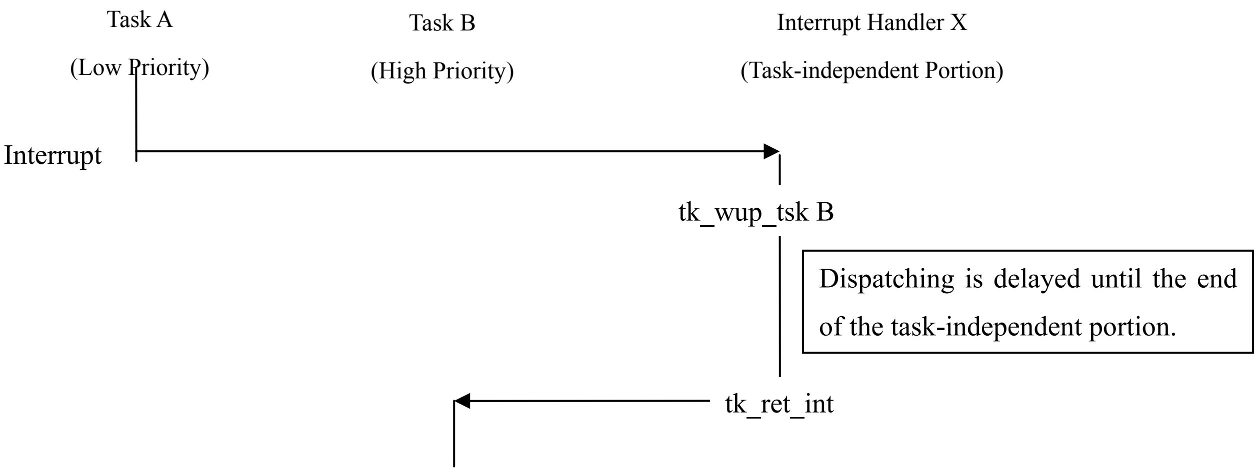


Figure 10(a): Delayed Dispatching in an Interrupt Handler

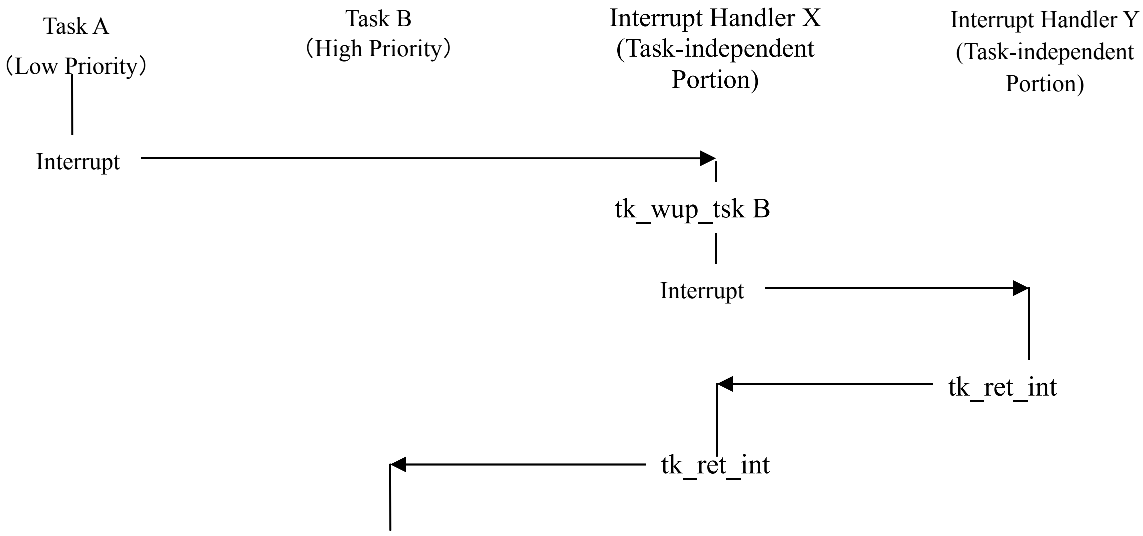


Figure 10(b): Delayed Dispatching in Interrupt Handlers (Interrupt Nesting)

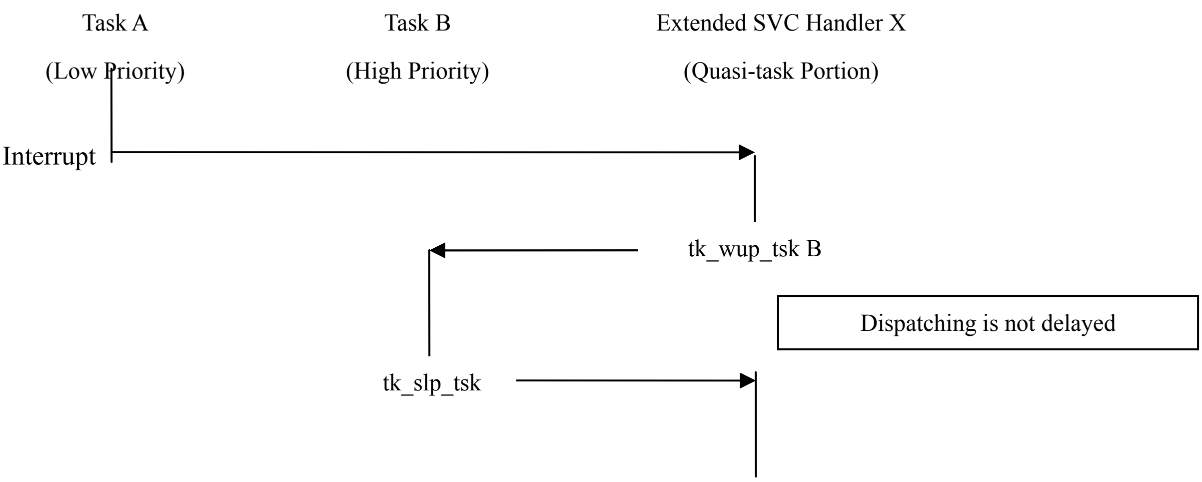


Figure 10(c): Dispatching in the Quasi-task Portion

2.5 Objects

"Object" is the general term for resources handled by T-Kernel. Besides tasks, objects include memory pools, semaphores, event flags, mailboxes and other synchronization and communication mechanisms, as well as time event handlers (cyclic handlers and alarm handlers).

In SMP T-Kernel, domains are newly added to objects of the T-Kernel 1.00 Specification.

Table 2 shows the list of SMP T-Kernel objects.

Table 2 List of Kernel Objects in SMP T-Kernel

Classification	Kernel Object
Task-Related	Tasks
Synchronization/Communication -Related	Semaphores Event Flags Mailboxes
Extended Synchronization/Communication -Related	Mutex Message Buffers Rendezvous Ports
Memory Pool-Related	Fixed Length Memory Pools Variable-size Memory Pools
Time Management-Related	Cyclic Handler Alarm Handler
Domain-Related	Domains

Each object is identified by an ID number. In the T-Kernel, the ID number cannot be specified by users and is automatically allocated when an object is created. For this reason, the ID number of the object must be dynamically acquired during execution of the program. In SMP T-Kernel, when each object is created, the object name can be specified. From this object name, the ID number can be acquired by using the domain function.

Object names are character strings of eight characters or less. One character is 1 byte long and when the name length is less than 8 bytes, 0s are used to fill up the remaining bytes. The usable characters are from a-z, A-Z, and 0-9 but SMP T-Kernel does not check the character-code value.

Attributes can generally be specified when an object is created. Attributes determine differences in object behavior or the initial object state. When TA_XXXXX is specified for an object, that object is called a "TA_XXXXX attribute object". If there is no particular attribute to be defined, TA_NULL (=0) is specified. Generally, there is no interface provided for reading attributes after an object is created.

In the value of an object or handler, the lower bits indicate system attributes and the upper bits indicate implementation-dependent attributes. This specification does not define the bit position at which the upper and lower distinction is to be made. In principle, however, the system attribute portion is assigned from the least significant bit (LSB) toward the most significant bit (MSB), and implementation-dependent attributes from the MSB toward the LSB. Bits not defining any attribute must be cleared to 0.

In some cases, an object may contain extended information. Extended information is specified when the object is created. Information passed in parameters when an object starts execution has no effect on T-Kernel behavior.

Extended information can be read by calling a system call to refer to the status of the object.

2.6 Memory

2.6.1 Address Space

Memory address space of T-Kernel 1.00 Specification is distinguished as system space (shared space) or task space (user space).The handling of memory in SMP T-Kernel also is the same as that of the T-Kernel 1.00 Specification.

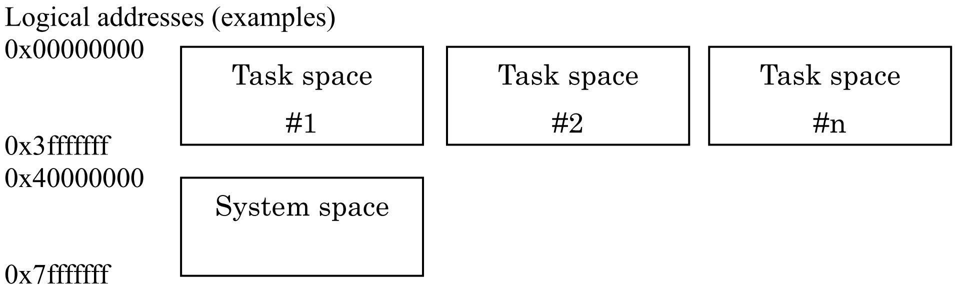


Figure 11: Address Space

Each address space is as follows.

(1) Task Space
Task space is accessible only from tasks belonging to this space. Multiple tasks can belong to one task space.

(2) System Space
System space is accessible from all tasks.

Due to hardware limitations, the arrangement of spaces in the logical address space is implementation-defined. However, as much as possible, they should be arranged from low addresses as in the following order.

(Low) Task space < System space (High)

Task-independent portions such as interrupt handlers belong to the task space of tasks executing immediately prior to entering the task-independent portion. This is the same as the task space of the currently running task returned with tk_get_tid. When there is no task in RUN state, task space is indeterminate.

T-Kernel does not create or manage address space. Normally, the creation and management of address spaces, etc. is realized by subsystems which manage the virtual memory, etc. of upper level systems such as Extensions.

All memory (system memory) where kernel can dynamically allocate is managed by the system memory management function. Memory used within the kernel and task stacks, message buffers, and memory pools are also allocated here. The target of system memory management is the memory of the system space. The memory management of task space is not performed in the kernel.

In a system with no MMU (or not using an MMU), it is assumed that task space does not exist. Likewise, task spaces do not exist in systems where only a single logical address space exists in each processor even if MMU is used.

2.6.2 Resident Memory and Nonresident Memory

Memory may be resident or nonresident. For nonresident memory, data is copied to memory from external storage such disks by accessing that memory. Therefore complicated processing such as disk access by a device driver is required. In T-Kernel, only declaring memory as resident and nonresident is performed and copying from disks, etc. by accessing nonresident memory is not performed. Normally, copying from disks, etc. is realized by subsystems which manage the virtual memory, etc. of upper level systems such as Extensions.

Accordingly, when nonresident memory is accessed, the device driver, etc. must be in operational state. Access is not possible during dispatch disabled or interrupts disabled state, or while a task-independent portion is executing. Similarly, in T-Kernel internal processing, it is necessary to avoid accessing nonresident memory in critical sections. One such case would be when the memory address passed in a system call parameter points to nonresident memory. Whether or not system call parameters are allowed to be in nonresident memory is an implementation-dependent matter.

In a system that does not use virtual memory, system call parameters or the like pointing to nonresident memory can be treated as being in resident memory by treating all memory spaces as resident.

2.7 Protection Levels

Protection levels in T-Kernel are set to the task and memory area. The SMP T-Kernel protection level specification is the same as that of the T-Kernel 1.00 Specification.

There are four protection levels from level 0 to level 3 with lower numbers indicating higher privilege. Tasks having protection level N can access the memory area of protection level N or higher. For example, a task with protection level 2 can access memory with protection levels 2 and 3.

A non-task portion (task-independent portion, quasi-task portion, etc.) runs at protection level 0. Only a task portion can run at protection levels 1 to 3. A task portion can also run at protection level 0.

Changing from one protection level to another is accomplished by invoking a system call or extended SVC, or by interrupt or CPU exception.

Usage of protection levels is defined as shown in Table 3.

[Table 3] Protection Level

Protection Level	Use
0	OS, Subsystem, Device Drivers etc.
1	(Reserved)
2	(Reserved)
3	Application Program Task

Reserved protection levels will be used by T-Kernel Extension, etc.

By using protection levels properly, access to memory areas can be prohibited even in the shared memory area in system space. As a result, destruction of the memory area content of the system by applications can be prevented, for example.

The protection level has been implemented by using functions of hardware such as CPU and MMU. Thus, the realized function of the protection level depends on hardware. For example, some MMUs support only two protection levels, privileged and user levels. In such a case, protection levels 0 to 2 are assigned to hardware-supported privileged level, and protection level 3 to the user level, as if there were four hardware-supported levels. In this case, level 3 is allocated to the user mode and level 0 to 2 are allocated to the privileged mode. There is no run-time distinction among level 0 to 2. In a system with no MMU, all protection levels 0 to 3 are treated as identical.

The protection level of a task has the following functions in addition to memory access restriction.

- Restrictions on system calls
  System calls cannot be used in protection levels that have a lower privilege than the set protection level (refer to Section 3.2.3).
- Access protection restrictions to kernel objects
  Access protection is ignored for tasks that have the same or higher privilege than the set protection level (refer to Section 2.8.4).
- Task exception restrictions
  Task exceptions cannot be used for tasks at protection level 0.

[Additional Notes]

Although the protection level was described in the "Memory" chapter in the T-Kernel 1.00 Specification, related functions are described in an independent chapter because it has the call restrictions on system calls and the restrictions on access protection to the object, etc. in addition to access restrictions on memory.

2.8 Domains

2.8.1 Concept of Domain

Domains show the logical location where kernel objects reside. Domains are functions introduced in MP T-Kernel and are not in T-Kernel 1.00 Specification

We can specify a domain when we create kernel object. The created object belongs to the specified domain. Kernel objects always belong to any one of the domains. If the domain is not specified when a kernel object is created, the kernel domain to be described later is automatically selected as default domain.

Domains realize the following functions.

(1) Search function for the kernel object ID number

Kernel objects can search for ID numbers from the object name specified at creation and the domain to which the kernel objects belong.

(2) Access protection function provision to kernel objects

Access protection on kernel objects is performed on operations from other objects according to the protection attribute specified during creation.

For example, objects with ‘`private`’ attribute can be operated only by the tasks that belong to the same domain.

[Rationale for the Specification]

In AMP or the combination model of AMP and SMP, a framework that shows to which processor (kernel) the kernel object belongs is necessary. Moreover, a function that acquires the ID number of kernel objects of other processors and the access protection function for kernel objects between processors are also necessary. Domains are introduced as the framework to put together these functions and to show where a kernel object resides.

The concept of a domain is effective for creating groups of kernel objects not only between processors or between kernels but also in large software systems.

For example, in the T-Kernel 1.00 Specification, some methods such as sharing the ID number as a variable or performing some kind of communication are necessary in order to pass the dynamically allocated ID number to other tasks. In large software systems, it is preferable that there be a regular method for acquiring the ID number of target objects. Here, when the search function for ID numbers by object name was discussed for draft specification, the duplication of names becomes a problem. For individual objects, assigning unique names inside a large system is difficult and it nullifies the advantage of assigning ID numbers automatically. Consequently, by using the domain as a name space to show where the name is valid, assigning names freely becomes possible for each processor or units of software components and middleware that comprise an application.

Based on these examinations, domain in MP T-Kernel was introduced as a general feature.

2.8.2 Kernel Domains and Hierarchical Structure of Domains

Domains are treated as kernel objects. Therefore, domains are created and deleted by system calls like other objects and are identified by ID numbers.

However, kernel domain is an exception. Kernel domains are created during system initialization. In SMP T-Kernel, one kernel domain exists in the system. The kernel domain is created when SMP T-Kernel is initialized and cannot be deleted.

Since a domain itself is an object, it belongs to one of the other domains. Therefore, domains form a hierarchical tree structure as shown in Figure 7. However, kernel domains do not belong to other domains since they are created initially at the same time. Consequently, a kernel domain is considered as belonging to itself. Kernel domains are also the roots in the hierarchical structure of domains.

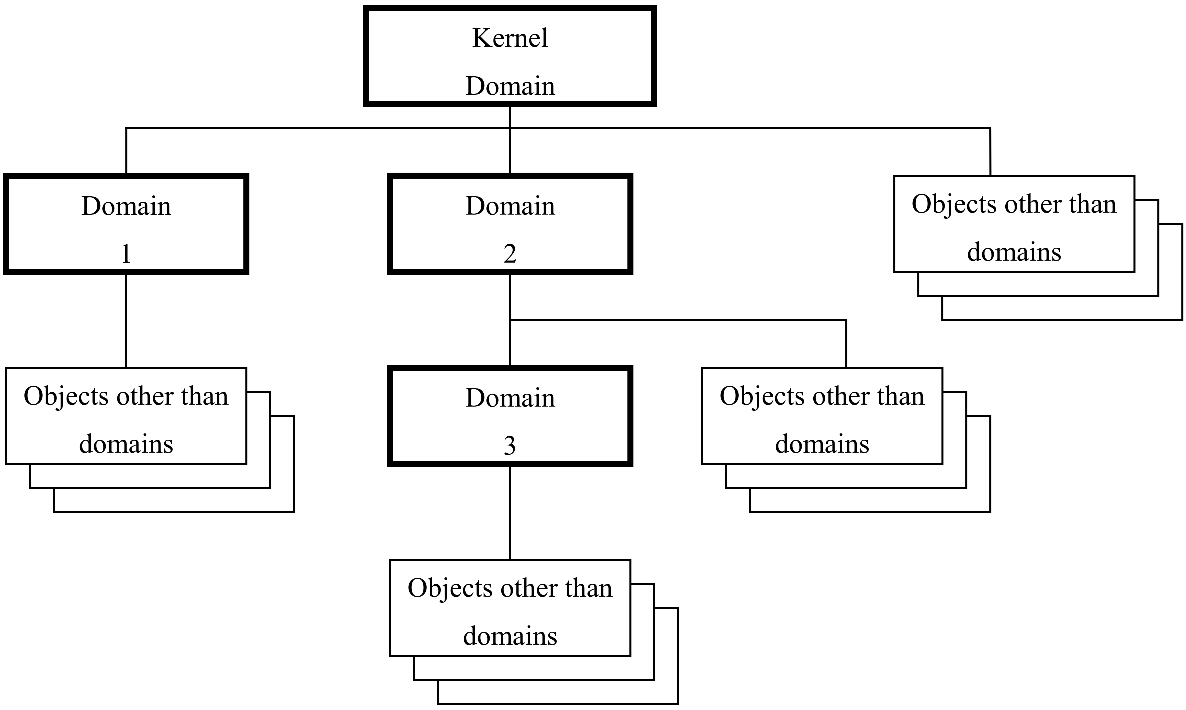


Figure 12 Hierarchical Structure of Domains

[Additional Notes]
In SMP T-Kernel, only one kernel domain exists in the entire system. However, in AMP T-Kernel, a kernel domain exists in each AMP T-Kernel instance allocated to each processor. Seen from the application, the kernel domain has 1-to-1 mapping to each MP T-Kernel instance and can show in which MP T-Kernel instance a kernel object resides.

2.8.3 ID Number Search Function

In T-Kernel, the object ID number is dynamically allocated when the object is created. Therefore, the ID number is not statically known in a program. The domain provides a function to search for the ID number of the object which belongs to the domain.

Object names can be specified when objects are created. Object names must be unique among the objects of the same type inside a domain to which the object belongs. For example, the same object name cannot be used for different semaphores belonging to one domain.

For a given object, the ID number of the object can be searched by using the domain, to which the object belongs, and the object name. Specifically, the ID number of the corresponding object can be acquired as a return value by passing the domain ID and the object name as arguments to system call tk_fnd_XXX (the XXX part changes for each object type).

Omitting the object name specification is also possible during object creation. In this case, the object does not have an object name. Objects without object names cannot use the ID number search function.

2.8.4 Domains and Access Protection Attributes

Access protection attributes can be specified when objects are created. Access protection provides restrictions on the operations of an object according to the access protection attribute specified for the domain to which the object belongs and the object itself, and realizes a protection function for an object.

The access protection attributes which can be specified for objects are the following types.

(1) `Private` Attribute

`Private` attribute objects can only be accessed from tasks and handlers that belong to the same domain as itself. Access from objects belonging to other domains is not possible.

(2) `Protected` Attribute

`Protected` attribute objects can only be accessed from tasks and handlers that belong to domains including the domains to which the object belongs and domain below in the domain hierarchy.

`Protected` attribute objects that belong to kernel domains are accessible from all tasks and handlers in the SMP T-Kernel.

(3) `Public` Attribute

`Public` attribute do not receive access restrictions according to the domain. Access is possible from all tasks and handlers.

Objects that have not specified access protection attributes have public attributes by default.

Access protection attributes can only be specified when objects are created, and cannot be dynamically changed.

The domains have `public` attribute by definition, and access protection attributes cannot be specified.

When an object is created, objects that will not be accessible due to the protection attribute from the context that issues the creation system call cannot be created. For example, `private` attribute objects cannot be created in domains which are different from the domain to which the invoking task belongs. The case where access is possible due to the exceptions from the protection levels to be described later is not included. Moreover, if access is possible, the creation of objects in other domains is possible.

[Additional Notes]

The purposes of access protection are the following two points.

First, the internal portion of domains is concealed and objects are protected from erroneous access. In large applications, programs are divided into components and modules, and it is important to conceal the internal portions of modules. The creation of object groups that comprise an application is possible due to the introduction of domains. By adding the access protection function to this group, concealing the internal portion of the domain, and dividing the program into modules becomes possible. However, erroneous operations are what the access protection function catches. Protection from malicious programs, or security function in other words, is not supported.

Second, implementation can be optimized by clarifying the range of access to objects.

[Rationale for the Specification]

The followings are reasons why the purpose of access protection is only protection from erroneous operations and why the security is not supported against malicious programs.

Although the function of security is important, the access protection of objects alone is inadequate for supporting security. Even for many other functions, modification is necessary, and this could end up with harming compatibility with the T-Kernel 1.00 Specification (compatibility may leave security holes). Such a change goes beyond the study of the MP T-Kernel Specification and disagrees with the design policy of MP T-Kernel which values compatibility with the T-Kernel 1.00 Specification. Therefore, security is handled outside the MP T-Kernel specifications.

2.8.5 Target and Restrictions of Access Protection

Access protection restricts access to object itself from others. It does not restrict access from object itself to others. For example, the task with `private` attribute does not allow access from tasks belonging to other domains, but it can access public attribute objects belonging to other domains.

That being said, the following are exceptions.

(1) Debugging Support Function

The debugger-support function provided by T-Kernel/DS acquires an internal state of an OS, traces the execution for the debugger, and must access all objects. However, use by general applications and middleware is prohibited as a rule. Therefore, the debugging support function is excluded from general access protection.

(2) Restrictions According to the Protection Level

Access protection can be restricted according to the protection level. For system calls from tasks and handlers at the specified protection level or greater, access protection becomes null.

For example, when the access privilege is null for system calls from protection level 1 or smaller, access by a system call from protection levels 0 and 1 becomes possible regardless of the access protection attribute of the object.

Such privileged access is used to access all objects from SMP T-Kernel and interrupt handlers, and upper level systems (Extensions and debuggers, etc.).

When an extended SVC handler is called from the task, the extended SVC handler is executed at protection level 0 but restrictions are maintained according to the protection level of the access protection of the task. That is, access to the object from the quasi-task portion is restricted by the protection level of the task portion that called it.

The protection level that restricts access is set by the system configuration information management function.

[Additional Notes]
Handlers are executed as the non-task portion. Since the protection level of the non-task portion is 0, the access protection becomes null. Consequently, handlers can access all objects.

2.9 Interrupt and Exception

2.9.1 Interrupt Handling

Interrupts in the T-Kernel specification include both external interrupts from devices and interrupts due to CPU exceptions. One interrupt handler can be defined for each interrupt number. Interrupt handlers can be designed for starting directly, (basically without OS intervention), or for starting via a high-level language support routine.

The interrupt processing of SMP T-Kernel is the same as that of the T-Kernel 1.00 Specification. Interrupt processing is executed by the processor where the interrupt occurs as a hardware function. Therefore, the interrupt handler runs on the processors where the interrupt occurs.

Since the correspondence of external interrupts and processors largely depends on hardware and system design, it is not specified in SMP T-Kernel and is implementation-defined.

2.9.2 Task Exception Handling

The T-Kernel specification defines task exception handling functions as dealing with exceptions. Note that exceptions other than those in the CPU are treated as interrupts.

A task exception handling function invokes a system call requesting task exception handling. It interrupts execution of a task, and runs a task exception handler. Execution of the task exception handler takes place in the same context as the interrupted task. Upon return from the task exception handler, the interrupted processing continues.

One task exception handler per task can be registered with an application.

2.10 Low-level Operation Function

In MP T-Kernel, the function that operates hardware directly or at a level close to hardware is called low-level operation function.

It is difficult to share low level operation functions because hardware-dependency is high, and functions are different in each system. However, providing a standard API specification is preferable from the standpoint that it improves software portability and readability. Consequently, although the standard API specification is provided in MP T-Kernel, the details are implementation-defined. Implementation that closely matches the standard specification as much as possible is requested but in the event there are difficulties in implementing the specification due to circumstances unique to the system, not implementing the API is permitted. Moreover, when a function that has not been implemented is called, if there is no problem with ignoring the call, it is requested so that execution of the application continues instead of system's returning an error in order to improve the portability of software. However, if ignoring the request affects the operation, E_NOSPT shall be returned.

The following functions of TK/SM are examples of the low-level operation functions.

- Address Space Management Function
- Interrupt Management Function
- I/O Port Access Support Function
- Inter-processor Management Function

[Additional Notes]

One of the roles of an OS is to abstract hardware and to conceal the details. However, the operation at a level close to hardware is necessary in embedded systems. Consequently, in MP T-Kernel, the low-level operation functions are provided.

The low-level operation functions use API standard, but a main part of the functions is implementation-dependent. Therefore, when software is ported, low level functions of the particular implementation of MP T-Kernel must be checked carefully.

Generally, applications that conduct low-level operations which need to be aware of system software and hardware such as device drivers and subsystems should use standard functions. It is preferable not to use low level functions from the standpoint of portability of upper-level applications. If low-level functions are to be used, adequate attention needs to be paid.

[Rationale for the Specification]

In MP T-Kernel, when the cache management function and spin lock control function were added, a framework for low level operation functions was created in addition to the interrupt management function and the I/O port access support function of the existing T-Kernel specification. Since these functions operate hardware directly or at a level close to hardware as described in the text, even if the API is standardized, change is necessary prior to porting application to different hardware. The purpose of the specification of low-level operation function is clarification and attracting attention.

Chapter 3 Common SMP T-Kernel Specifications

3.1 Data Types

3.1.1 General Data Types

The general data types used in SMP T-Kernel are described as follows. These data types are common with the T-Kernel 1.00 Specification.

```
typedef char            B;              /* signed 8-bit integer */
typedef short           H;              /* signed 16-bit integer */
typedef int             W;              /* signed 32-bit integer */
typedef unsigned char   UB;             /* unsigned 8-bit integer */
typedef unsigned short  UH;             /* unsigned 16-bit integer */
typedef unsigned int    UW;             /* unsigned 32-bit integer */

typedef char            VB;             /* 8-bit data without a fixed type */
typedef short           VH;             /* 16-bit data without a fixed type */
typedef int             VW;             /* 32-bit data without a fixed type */
typedef void            *VP;            /* pointer to data without a fixed type */

typedef volatile B      _B;             /* volatile declaration */
typedef volatile H      _H;
typedef volatile W      _W;
typedef volatile UB     _UB;
typedef volatile UH     _UH;
typedef volatile UW     _UW;

typedef int             INT;            /* signed integer of processor bit width */
typedef unsigned int    UINT;           /* unsigned integer of processor bit width */

typedef INT             ID;             /* ID in general*/
typedef INT             MSEC;           /* time (milliseconds) in general    */

typedef void            (*FP)();        /* function address in general */
typedef INT             (*FUNCP)();     /* function address in general */

#define LOCAL           static          /* local symbol definition */
#define EXPORT                          /* global symbol definition */
#define IMPORT          extern          /* global symbol reference */

/*
 * Boolean  values
 *          TRUE = 1 is defined below, but any value other than 0 is TRUE.
 *          A decision such as bool == TRUE must be avoided for this reason.
 *          Instead use bool != FALSE.
 */
typedef INT             BOOL;
#define TRUE            1               /* True */
#define FALSE           0               /* False */

/*
 * TRON character codes
 */
typedef UH              TC;             /* TRON character code */
#define TNULL           ((TC)0)         /* TRON code string termination */
```

[Additional Notes]

VB, VH, and VW differ from B, H, and W in that the former mean only the bit width is known, not the contents of the data type, whereas the latter clearly is used for integer type.

Processor bit width must be 32 bits or above. INT and UINT must therefore always have a width of 32 bits or more.

Parameters such as stksz, wupcnt, and message size that clearly do not take negative values are also in principle signed integer (INT, W) data type. This is in keeping with the overall TRON rule that integers should be treated as signed numbers as much as possible. As for the timeout (TMO tmout) parameter, its being a signed integer permits the use of TMO_FEVR (= -1) having special meaning. Parameters with unsigned data type are those treated as bit patterns (object attribute, event flag, etc.).

3.1.2 Other Defined Data Types

The following names are used for other data types that appear frequently or have special meaning, in order to make the parameter meaning clear.

```
typedef INT         FN;         /* Function code */
typedef INT         RNO;        /* Rendezvous number */
typedef UINT        ATR;        /* Object/handler attributes */
typedef INT         ER;         /* Error code */
typedef INT         PRI;        /* Priority */
typedef INT         TMO;        /* Timeout*/
typedef UINT        RELTIM;     /* Relative time */

typedef struct systim {         /* System time */
        W       hi;             /* High 32 bits */
        UW      lo;             /* Low 32 bits */
} SYSTIM;

/*
 * Common constants
 */
#define NULL        0           /* Null pointer */
#define TA_NULL     0           /* No special attributes indicated */
#define TMO_POL     0           /* Polling */
#define TMO_FEVR    (-1)        /* Eternal Wait */
```

A data type that combines two or more data types is represented by its main data type. For example, the value returned by tk_cre_tsk can be a task ID or error code, but since it is mainly a task ID, the data type is ID.

3.2 System Calls

The basic specification such as the format for system calls of SMP T-Kernel conforms to the T-Kernel 1.00 Specification. There is only a minor difference between the T-Kernel 1.00 Specification and the content described in this chapter.

3.2.1 System Call Format

T T-Kernel adopts C as the standard high-level language, and standardizes interfaces for system call execution from C language routines.

On the other hand, the interface for system call execution from assembly language is implementation-defined. Calling by means of a C language interface is recommended even when an assembly language program is created. In this way, portability is assured for programs written in assembly language even if the OS changes, as long as the CPU is the same.

The following common rules are established for system call interfaces.

- All system calls are defined as C functions.
- A function return code of 0 or a positive value indicates normal completion, while negative values are used for error codes.

All system call interfaces are provided as libraries. C language macros, in-line functions and in-line assembly code are not used. The reason is that C macros and in-line functions can be used only from a C program. Moreover, since in-line functions and in-line assembly are not standard C features, their functioning is in many cases compiler-dependent, diminishing portability.

3.2.2 System Calls Possible from Task-Independent Portion and Dispatch Disabled State

Even when the following system calls are issued from the task-independent portion or dispatch disabled state, the system calls must operate in the same way as if they were issued from the task portion (E_CTX must not be returned either).

tk_sta_tsk	Start task
tk_wup_tsk	Wakeup task
tk_rel_wai	Release wait
tk_sus_tsk	Suspend task
tk_sig_sem	Signal semaphore
tk_set_flg	Set event flag
tk_rot_rdq	Rotate task queue
tk_get_tid	Get task ID
tk_sta_cyc	Start cyclic handler
tk_stp_cyc	Stop cyclic handler
tk_sta_alm	Start alarm handler
tk_stp_alm	Stop alarm handler
tk_ref_tsk	Reference task status
tk_ref_cyc	Reference cyclic handler status
tk_ref_alm	Reference alarm handler status
tk_get_prc	Get executing Processor ID
tk_ref_sys	Reference system status
tk_ret_int	Return from interrupt handler

Even when the following system calls are issued from the dispatch disabled state, the system calls must operate in the same way as if they were issued from the dispatchable state (E_CTX must not be returned either).

tk_fwd_por	Forward rendezvous to Rendezvous port
tk_rpl_rdv	Reply Rendezvous

Operations of system calls other than the above when they are issued from the task-independent portion and the dispatch disabled state are implementation-defined.

3.2.3 Restricting System Call Invocation

The protection levels at which a system call is invokable can be restricted. In this case, if a system call is issued from a task (task portion) running at lower than the specified protection level, the error code E_OACV is returned. Extended SVC calling cannot be restricted.

If, for example, issuing system call from a protection level lower than 1 is prohibited, system calls cannot be invoked from tasks running at protection levels 2 and 3. Tasks running at those levels will only be able to make extended SVC calls, and are programmed using subsystem functions only.

This kind of restriction is used when T-Kernel is combined with T-Kernel Extension, to prevent tasks based on T-Kernel Extension specification from directly accessing T-Kernel functions. It allows T-Kernel to be used as a micro-kernel.

The protection level restriction on system call invocation is set using the system configuration information management functions (see 5.7).

3.2.4 Modifying a Parameter Packet

Some parameters passed to system calls use a packet format. The packet format parameters are of two kinds, either input parameters passing information to a system call (e.g., T_CTSK) or output parameters returning information from a system call (e.g., T_RTSK).

Additional information that is implementation-dependent may be added to a parameter packet. However, changing the data types and the order of information defined in the standard specification or deleting any of this information is not allowed. When implementation-dependent information is added, it must be placed after the standard defined information.

When implementation-dependent information is added to a packet of input information passed to a system call (T_CTSK, etc.), if the system call is invoked while this additional information is not yet initialized (memory contents indeterminate), the system call must still function normally.

Ordinarily a flag indicating that valid values are set in the additional information is defined in the implementation-dependent area of attribute flag included in the standard specification. When that flag is set (1), the additional information is to be used; and when the flag is not set (0), the additional information is not initialized (memory contents indeterminate) and the default settings are to be used.

The reason for this specification is to ensure that a program developed within the scope of the standard specification will be capable of running on an OS with implementation-dependent functional extensions, simply by recompiling.

3.2.5 Function Codes

Function codes are numbers assigned to each system call and used to identify the system call.

Although specific values for the function code of system calls are implementation-defined, different negative values are allocated in each respective system call.

Positive values are allocated to the function code of extended SVC.

3.2.6 Error Codes

System call return codes are in principle to be signed integers. When an error occurs, a negative error code is returned; and if processing is completed normally, E_OK (= 0) or a positive value is returned. The meaning of returned codes in the case of normal completion is specified separately for each system call. An exception to this principle is that there are some system calls that do not return when called. A system call that does not return is declared in the C language API as having no return code (that is, a void type function).

An error code consists of the main error code and sub error code. The low 16 bits of the error code are the sub error code, and the remaining high bits are the main error code. T-Kernel/OS does not use a sub error code, the low bits are always 0.

The following macros are prepared for the conversion of error codes, main error codes, and sub error codes.

```
#define MERCD(er)          ( (ER)(er) >> 16 )              /* Main error code */
#define SERCD(er)          ( (H)(er) )                     /* Sub error code */
#define ERCD(mer, ser)     ( (ER)(mer) << 16 | (ER)(UH)(ser) )
```

Main error codes are classified into error classes based on the necessity of their detection, the circumstances in which they occur and other factors. For more details on error codes and error classes, refer to 5.2 Error Code List.

3.2.7 Timeout

A system call that may enter WAIT state has a timeout function. If processing is not completed by the time the specified timeout interval has elapsed, the processing is canceled and the system call returns.

A system call returns an E_TMOUT when it returns due to a timeout. In accordance with the principle that there should be no side-effects from invoking a system call if that system call returns an error code, the invocation of a system call that times out should in principle result in no change in system state. An exception to this is when the functioning of the system call is such that it cannot return to its original state if processing is canceled. This is indicated in the system call description.

If the timeout interval is set to 0, a system call does not enter WAIT state even when a situation arises in which it would ordinarily go to WAIT state. In other words, a system call with timeout set to 0 when it is invoked has no possibility of entering WAIT state. Invoking a system call with timeout set to 0 is called polling; that is, a system call that performs polling has no chance of entering WAIT state.

The descriptions of individual system calls, as a rule, describe the behavior when there is no timeout (in other words, when an eternal wait occurs). Even if the system call description says that the system call “enters WAIT state” or “is put in WAIT state”, if a timeout is set and that time interval elapses before processing is completed, the WAIT state is released and the system call returns error code E_TMOUT. In the case of polling, the system call returns E_TMOUT without entering WAIT state.

Timeout (TMO type) may be a positive integer, TMO_POL (= 0) for polling, or TMO_FEVR (=−1) for eternal wait. If a timeout interval is set, the timeout processing must be guaranteed to take place after the specified interval from the system call issuing has elapsed.

[Additional Notes]

Since a system call that performs polling does not enter WAIT state, there is no change in the precedence of the task calling it.

In a general implementation, when the timeout is set to 1, timeout processing takes place on the second time tick after a system call is invoked. Since a timeout of 0 cannot be specified (0 being allocated to TMO_POL), in this kind of implementation timeout does not occur on the initial time tick after the system call is invoked.

3.2.8 Relative Time and System Time

When the time of an event occurrence is specified relative to another time, such as the time when a system call was invoked, relative time (RELTIM type) is used. If relative time is used to specify event occurrence time, it is necessary to guarantee that event processing will take place after the specified time has elapsed from the time base. Aside from event occurrence, relative time (RELTIM type) is also used for cases such as the interval between two event occurrences. In such cases, the method of interpreting the given relative time is specified for each case.

When time is specified as an absolute value, system time (SYSTIM type) is used. The T-Kernel specification provides a function for setting system time, but even if the system time is changed using this function, there is no change in the real world time (actual time) at which an event, which was specified using relative time, occurs. What changes is the system time at which an event occurs that was specified using relative time.

- SYSTIM System time

Time resolution 1 millisecond, 64-bit signed integer

```
typedef struct systim {
        W       hi;     /* high 32 bits */
        UW      lo;     /* low 32 bits */
} SYSTIM;
```

- RELTIM Relative time

Time resolution 1 millisecond, 32-bit or longer unsigned integer (UW)

```
typedef UW      RELTIM;
```

- TMO Timeout time

Time resolution 1 millisecond, 32-bit or longer signed integer (W)

```
typedef W       TMO;
```

Eternal wait can be specified as TMO_FEVR = (−1).

[Additional Note]

Timeout or other such processing must be guaranteed to occur after the time specified as RELTIM or TMO has elapsed. For example, if the timer interrupt cycle is 1 ms and a timeout of 1 ms is specified, timeout occurs on the second timer interrupt.

(At the first timer interrupt, the elapsed time does not exceed 1 ms.)

3.3 High-Level Language Support Routines

In T-Kernel, high-level language support routine is provided so that even if a task or handler is written in a high-level language, the kernel-related processing can be kept separate from the language environment-related processing. Whether or not high-level language support routine is used is specified as either the object attribute or the handler attribute (TA_HLNG). High-level language support routine in SMP T-Kernel is the same as that of the T-Kernel 1.00 Specification.

When TA_HLNG is not specified, a task or handler is started directly from the start address passed in a parameter to tk_cre_tsk or tk_def_???; whereas when TA_HLNG is specified, first the high-level language startup processing routine (high-level language support routine) is started, then from this routine an indirect jump is made to the task start address or handler address passed in a parameter to tk_cre_tsk or tk_def_???. Viewed from the OS, the task start address or handler address is a parameter passed to the high-level language support routine. Separating the kernel processing from the language environment processing in this way facilitates support for different language environments.

Use of high-level language support routine has the further advantage that when a task or handler is written as a C language function, a system call for task exit or return from a handler can be executed automatically, simply by performing a function return (return or “)”).

In a system that uses an MMU, however, whereas it is relatively easy to implement a high-level language support routine in the case of an interrupt handler or the like that runs at the same protection level as the OS, it is more difficult in the case of a task or task exception handler running at a different protection level from that of the OS. For this reason, when a high-level language support routine is used for a task, there is no guarantee that the task will exit by a return from the function. In the case of a task exception handler, the high-level language support routine is supplied as source code and is to be embedded in the user program.

The internal working of a high-level language support routine is as illustrated in Figure 13.

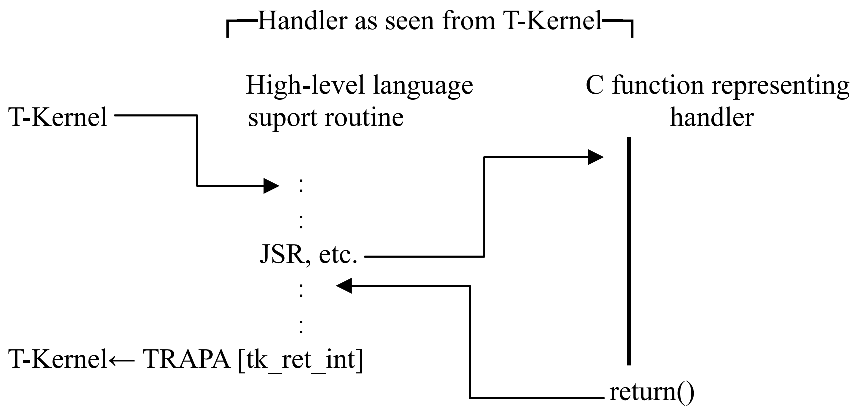


Figure 13: Behavior of High-Level Language Support Routine

Chapter 4 SMP T-Kernel/OS Functions

This chapter describes in detail the system calls provided by the SMP T-Kernel Operating System (T-Kernel/OS).
The following functions exist in SMP T-Kernel/OS.

- Task Management Functions
- Task-Dependent Synchronization Functions
- Task Exception Handling Functions
- Synchronization and Communication Functions
- Extended Synchronization and Communication Functions
- Memory Pool Management Functions
- Time Management Functions
- Domain Management Functions
- Interrupt Management Functions
- System Management Functions
- Subsystem Management Functions

4.1 Task Management Functions

Task management functions manipulate or refer to task states. Functions are provided for creating and deleting a task, for starting and exiting a task, changing task priority, and referring to task state. A task is an object identified by an ID number called a task ID. Task states and scheduling rules are explained in 2.3 Task States and Scheduling Rules.

For control of execution order, a task has a base priority and current priority. When simply “task priority” is talked about, this means the current priority. The base priority of a task is initialized as the startup priority when a task is started. If the mutex function is not used, the task current priority is always identical to its base priority. For this reason, the current priority immediately after a task is started is the task startup priority. When the mutex function is used, the current priority is set as discussed in "4.5.1 Mutex".

The kernel does not perform processing for freeing of resources acquired by a task (semaphore resources, and memory blocks, etc.) upon task exit, other than mutex unlocking. Freeing of task resources is the responsibility of the application.

In SMP T-Kernel, new members are added to the structure passed as parameter to tk_cre_tsk so that the designation of the domain and the access protection attribute to which the task belongs is possible when the task is created. Access protection is applied to system calls that specify task ID.

Service calls that are different from the T-Kernel 1.00 Specification are summarized in the table below. For more details, refer to explanation of each system call.

Call Name	Function	Different from T-Kernel 1.00 Specification
tk_cre_tsk	Create Task	×
tk_del_tsk	Delete Task	△
tk_sta_tsk	Start Task	△
tk_ext_tsk	Exit Task	○
tk_exd_tsk	Exit and Delete Task	○
tk_ter_tsk	Terminate Task	△
tk_chg_pri	Change Task Priority	△
tk_chg_slt	Change Task Slice-time	△
tk_get_tsp	Get Task Space	△
tk_set_tsp	Set Task Space	△
tk_get_rid	Get Task Resource ID	△
tk_set_rid	Set Task Resource ID	△
tk_get_reg	Get Task Registers	×
tk_set_reg	Set Task Registers	×
tk_get_cpr	Get Task Co-processor Registers	×
tk_set_cpr	Set Task Co-processor Registers	×
tk_inf_tsk	Get task statistics	△
tk_ref_tsk	Reference Task Status	△

Different from T-Kernel 1.00 Specification ○:No ×:Yes △: Different only in that E_DACV is returned due to the access protection

tk_cre_tsk: Create Task

[C Language Interface]

```
ID tskid = tk_cre_tsk ( T_CTSK *pk_ctsk ) ;
```

[Parameters]

T_CTSK*	pk_ctsk	Packet of Create Task	Information about the task to be created

pk_ctsk detail:

VP	exinf	Extended Information	
ATR	tskatr	Task Attribute	
FP	task	Task Start Address	
PRI	itskpri	Initial Task Priority	
INT	stksz	Stack Size (bytes)	
INT	sstksz	SystemStackSize (bytes)	
VP	stkptr	UserStackPointer	
VP	uatb	Task space page table	
INT	lsid	LogicalSpaceID	
ID	resid	ResourceID	
ID	domid	DomainID	
UW	assprc	AssignProcessor	Execution processor specification
UB	oname[8]	Object name	

——(Other implementation-dependent parameters may be added beyond this point.)——

[Return Parameters]

ID	tskid	Task ID
	or	Error Code

[Error Codes]

E_NOMEM	Insufficient memory (memory for control block or user stack cannot be allocated)
E_LIMIT	Number of tasks exceeds the system limit
E_RSATR	Reserved attribute (tskatr is invalid or cannot be used), or the specified co-processor does not exist
E_NOSPT	Unsupported function (when TA_USERSTACK or TA_TASKSPACE is not supported)
E_PAR	Parameter error. pk_ctsk, task, stkptr, or itskpri is invalid
E_ID	Invalid resource ID (resid, or domid is invalid)
E_NOEXS	Object does not exist (domain of domid does not exist)
E_NOCOP	The specified co-processor cannot be used not installed in the currently running hardware, or abnormal co-processor operation was detected
E_ONAME	The specified object name has already been used

[Description]

Creates a task, assigning it to a task ID number. This system call allocates a TCB (Task Control Block) for the created task and initializes it based on itskpri, task, stksz and other parameters.

After the task is created, it is initially in the DORMANT state.

Task priority values are specified from 1 to 140, with smaller numbers indicating higher priority.

exinf can be used freely by the user to insert miscellaneous information about the task. The information set here is passed to the task as a startup parameter information and can be referred to by calling tk_ref_tsk. If a larger area is needed for indicating user information, or if the information may need to be changed after the task is created, it can be done by allocating separate memory for this purpose and putting the memory packet address in exinf. The OS pays no attention to the contents of exinf.

tskatr indicates system attributes in its low bits and implementation-dependent information in the high bits. The system attributes part of tskatr is as follows.

```
tskatr := (TA_ASM ‖ TA_HLNG)
    | [TA_SSTKSZ] | [TA_USERSTACK] | [TA_TASKSPACE] | [TA_RESID] | [TA_ONAME]
    | (TA_RNG0 ‖ TA_RNG1 ‖ TA_RNG2 ‖ TA_RNG3)
```

| [TA_ASSPRC]
| [TA_DOMID] | [(TA_PROTECTED || TA_PRIVATE || TA_PUBLIC)]
| [TA_COP0] | [TA_COP1] | [TA_COP2] | [TA_COP3] | [TA_FPU]

TA_ASM	Indicates that the task is written in assembly language
TA_HLNG	Indicates that the task is written in high-level language
TA_SSTKSZ	Specifies the system stack size
TA_USERSTACK	Points to the user stack
TA_TASKSPACE	Points to the task space
TA_RESID	Specifies the resource group to which the task belongs
TA_ONAME	Specifies DS Object name
TA_RNGn	Indicates that the task runs at protection level n
TA_ASSPRC	Specifies the execution processor
TA_DOMID	Specifies the domain to which the task belongs
TA_PROTECTED	Sets the access protection attribute to protect
TA_PRIVATE	Sets the access protection attribute to private
TA_PUBLIC	Sets the access protection attribute to public
TA_COPn	Specifies use of the nth co-processor (including floating point co-processor or DSP)
TA_FPU	Specifies use of a floating point co-processor (when a co-processor specified in TA COPn is a general-purpose FPU particularly for floating point processing and not dependent on the CPU)

The function for specifying implementation-dependent attributes can be used, for example, to specify that a task is subject to debugging.

```
#define TA_ASM          0x00000000   /* Assembly program */
#define TA_HLNG         0x00000001   /* High-level language program */
#define TA_SSTKSZ       0x00000002   /* System stack size */
#define TA_USERSTACK    0x00000004   /* User stack pointer */
#define TA_TASKSPACE    0x00000008   /* Task space pointer */
#define TA_RESID        0x00000010   /* Task resource group */
#define TA_ONAME        0x00000040   /* Object name */
#define TA_RNG0         0x00000000   /* Run at protection level 0 */
#define TA_RNG1         0x00000100   /* Run at protection level 1 */
#define TA_RNG2         0x00000200   /* Run at protection level 2 */
#define TA_RNG3         0x00000300   /* Run at protection level 3 */
#define TA_COP0         0x00001000   /* Use ID=0 coprocessor */
#define TA_COP1         0x00002000   /* Use ID=1 coprocessor */
#define TA_COP2         0x00004000   /* Use ID=2 coprocessor */
#define TA_COP3         0x00008000   /* Use ID=3 coprocessor */
#define TA_DOMID        0x00010000   /* Specify the domain */
#define TA_ASSPRC       0x00020000   /* Specify the execution processor */
#define TA_PRIVATE      0x00040000   /* Set the protection attribute to private */
#define TA_PROTECTED    0x00080000   /* Set the protection attribute to protect */
#define TA_PUBLIC       0x00000000   /* Set the protection attribute to public */
```

When TA_HLNG is specified, excecution of the task does not pass control to the task start address directly, but by passing it through a high-level language environment configuration program (high-level language support routine). The task takes the following form in this case.

```
void task( INT stacd, VP exinf )
{
        /*
                processing
         */
       tk_ext_tsk();or tk_exd_tsk(); /* Exit task */
}
```

The startup parameters passed to the task include the task startup code stacd specified in tk_sta_tsk, and the extended information exinf specified in tk_cre_tsk.

The task cannot (must not) be terminated by a simple return from the function, otherwise the operation will be indeterminate (implementation-dependent).
The form of the task when the TA_ASM attribute is specified is implementation-dependent, but stacd and exinf must be passed as startup parameters.
The task runs at the protection level specified in the TA_RNGn attribute. When a system call or extended SVC is called, the protection level goes to 0, then goes back to its original level upon return from the system call or extended SVC.
Each task has two stack areas, a system stack and user stack. The user stack is used at the protection level specified in TA_RNGn, while the system stack is used at protection level 0. When the calling of a system call or extended SVC causes the protection level to change, the stack is also switched. Note that a task running at TA_RNG0 does not switch protection levels, so there is no stack switching either. When TA_RNG0 is specified, the combined total of the user stack size and system stack size is the size of one stack, employed as both a user stack and system stack.
When TA_SSTKSZ is specified, sstksz is valid. If TA_SSTKSZ is not specified, sstksz is ignored and the default size is used.
When TA_USERSTACK is specified, stkptr is valid. In this case, a user stack is not provided by the OS, but must be allocated by the caller. stksz must be set to 0. If TA_USERSTACK is not specified, stkptr is ignored. Note that if TA_RNG0 is set, TA_USERSTACK cannot be specified.
When TA_TASKSPACE is specified, uatb and lsid are valid and are set as task space. If TA_TASKSPACE is not specified, uatb and lsid are ignored and task space is undefined. During the time task space is undefined, only system space can be accessed; access to task (user) space is not allowed. Whether or not TA_TASKSPACE was specified, task space can be changed after a task is created. Note that when task space is changed, in no case does it revert to the task space set at task creation, even when the task returns to DORMANT state, but the task always uses the most recently set task space.
When TA_RESID is specified, resid is valid and specifies the resource group to which the task belongs. If TA_RESID is not specified, resid is ignored and the task belongs to the system resource group. Note that if the resource group of a task is changed, in no case does it revert to the resource group set at task creation, even when the task returns to DORMANT state, but the task always retains the most recently set resource group (see tk_cre_res).
When TA_ASSPRC is specified, assprc is valid. The value of assprc is a bit pattern that indicates processors which can execute the task. The bit patterns are defined as follows.

```
#define TP_PRC1      0x00000001    Processor with Processor ID Number 1
#define TP_PRC2      0x00000002    Processor with Processor ID Number 2
#define TP_PRC3      0x00000004    Processor with Processor ID Number 3
#define TP_PRC4      0x00000008    Processor with Processor ID Number 4
                          ～
#define TP_PRC32     0x80000000    Processor with Processor ID Number 32
```

Multiple execution processors can be specified. For example, to specify the processors with ID number 1 and ID number 2, the value of the logical sum of TP_PRC1 and TP_PRC2 is specified. When TA_ASSPRC is not specified, all processors can execute the task. This is the same as when all bits of assprc are set.
When TA_DOMID is specified, domid is valid, and the domain of domid is set as the domain to which the task belongs. When TA_DOMID is not specified, domid is ignored and the task belongs to the domain to which the kernel domain belongs.
TA_PROTECTED, TA_PRIVATE, and TA_PUBLIC specify the access protection attribute of a task. When either of the access protection attributes is not specified, access protection is set to the public attribute. Tasks that invoking tasks cannot access due to access protection with the combination of the domain to which the task belongs and the access protection attribute cannot be created. When the corresponding specification is done, E_PAR is returned.
When TA_ONAME is specified, oname is valid and is set as the object name. When TA_ONAME is not specified, the object name is not set. The object name must be unique inside the domain to which the task belongs. When an object name that has already been used by other tasks is specified, E_ONAME is returned. When the length of the character string specified for oname is 0 (initial character is termination 0), the object name is not set regardless of the TA_ONAME specification.

[Additional Notes]
A task runs either at the protection level set in TA_RNGn or at protection level 0. For example, a task for which TA_RNG3 is specified in no case runs at protection level 1 or 2.
In a system with a separate interrupt stack, interrupt handlers also use the system stack. An interrupt handler runs at protection level 0.
The system stack default size is decided taking into account the amount taken up by system call execution and, in a system with separate interrupt stack, the amount used by interrupt handlers.
The system stack is resident memory in system space used at protection level 0. If TA_USERSTACK is not specified, the user stack is resident memory in system space used at the protection level specified in the TA_RNGn attribute. If TA_USERSTACK is specified, the user stack memory attributes are as specified by the caller of this system call. Task space may be made nonresident memory.
The definition of TA_COPn is dependent on the CPU and other hardware and is not portable. TA_FPU is provided as a

portable notation method only for the definition in TA_COPn of a floating point processor. If, for example, the floating point processor is TA_COP0, then TA_FPU = TA_COP0. If there is no particular need to specify the use of a co-processor for floating point operations, TA_FPU = 0 is set.

Even in a system without an MMU, for the sake of portability all attributes including TA_RNGn must be accepted. It is possible, for example, to handle all TA_RNGn as equivalent to TA_RNG0, but error must not be returned. In the case of TA_USERSTACK and TA_TASKSPACE, however, E_NOSPT may be returned, since there are many cases where these cannot be supported without an MMU.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- TA_DOMID, TA_PROTECTED, TA_PRIVATE, and TA_PUBLIC are added to the task attribute, and the domain to which the task belongs and the access protection attribute can be specified.
- TA_ASSPRC was added to the task attribute, and the execution processor can be specified.
- The DS Object name was abolished and replaced by the establishment of the object name. While the former was a name for debugging, the latter is a name which can be used in general for searching domain IDs, etc. The object name cannot use the existing same name with the same type of object in the same domain.

Delete Task tk_del_tsk

tk_del_tsk: Delete Task

[C Language Interface]

ER ercd = tk_del_tsk (ID tskid) ;

[Parameters]

ID	tskid	Task ID

[Return Parameters]

ER	ercd	Error Code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_OBJ	Invalid object state (the task is not in DORMANT state)
E_DACV	Access protection violation

[Description]

Deletes the task specified in tskid.

This system call changes the state of the task specified in tskid from DORMANT state to NONEXISTENT state (no longer exists in the system), releasing the TCB and stack area that were assigned to the task. The task ID number is also released. When this system call is issued for a task not in DORMANT state, error code E_OBJ is returned.

This system call cannot specify the invoking task. If the invoking task is specified, error code E_OBJ is returned because the invoking task is not in DORMANT state. The invoking task is deleted not by this system call but by the tk_exd_tsk system call.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences with the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to access protection, E_DACV is returned.

Start Task tk_sta_tsk

tk_sta_tsk: Start Task

[C Language Interface]

```
ER ercd = tk_sta_tsk ( ID tskid, INT stacd ) ;
```

[Parameters]

ID	tskid	Task ID
INT	stacd	Task Start Code

[Return Parameters]

ER	ercd	Error Code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_OBJ	Invalid object state (the task is not in DORMANT state)
E_DACV	Access Protection Violation

[Description]

Starts the task specified in tskid. This system call changes the state of the specified task from DORMANT state to READY state.

Parameters to be passed to the task when it starts can be set in stacd. These parameters can be referred to from the started task, enabling use of this feature for simple message passing.

The task priority when it starts is the task startup priority (itskpri) specified when the started task was created.

Start requests by this system call are not queued. If this system call is issued while the target task is in a state other than DORMANT state, the system call is ignored and error code E_OBJ is returned to the calling task.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.

tk_ext_tsk: Exit Task

[C Language Interface]

```
void tk_ext_tsk ( void ) ;
```

[Parameters]

None

[Return Parameters]

※ Does not return to the context issuing the system call.

[Error Codes]

※ The following error can be detected; but since this system call does not return to the context issuing the system call, even when an error is detected, an error code cannot be passed directly in a system call return parameter. The behavior in case an error occurs is implementation-dependent.

E_CTX Context error (issued from task-independent portion or in dispatch disabled state)

[Description]

Exits the invoking task normally and changes its state to DORMANT state.

[Additional Notes]

When a task terminates by tk_ext_tsk, the resources acquired by the task up to that time (memory blocks, semaphores, etc.) are not automatically freed. The user is responsible for releasing such resources before the task exits.

tk_ext_tsk is a system call that does not return to the context from which it was called. Even if an error code is returned when an error of some kind is detected, normally no error checking is performed in the context from which the system call is invoked, leaving the possibility that the program will hang. For this reason, these system calls do not return even if an error is detected.

As a rule, the task priority and other information included in the TCB is reset when the task returns to DORMANT state. If, for example, the task priority is changed by tk_chg_pri and later terminated by tk_ext_tsk, the task priority reverts to the startup priority (itskpri) specified when the task was started. It does not keep the task priority in effect at the time tk_ext_tsk was executed.

System calls that do not return to the calling context are those named tk_ret_??? or tk_ext_??? (tk_exd_???).

[Items Concerning SMP T-Kernel]

There is no difference from the T-Kernel 1.00 Specification.

Exit and Delete Task tk_exd_tsk

tk_exd_tsk: Exit and Delete Task

[C Language Interface]

```
void tk_exd_tsk ( void ) ;
```

[Parameters]

None.

[Return Parameters]

※ Does not return to the context issuing the system call.

[Error Codes]

※ The following error can be detected but since this system call does not return to the context issuing the system call, even when an error is detected, an error code cannot be passed directly in a system call return parameter. The behavior in case an error occurs is implementation-dependent.

E_CTX Context error (issued from task-independent portion or in dispatch disabled state)

[Description]

Terminates the invoking task normally and also deletes it. This system call changes the state of the invoking task to NON-EXISTENT state (no longer exists in the system).

[Additional Notes]

When a task terminates by tk_exd_tsk, the resources acquired by the task up to that time (memory blocks, and semaphores, etc.) are not automatically freed. The user is responsible for releasing such resources before the task exits.

tk_exd_tsk is a system call that does not return to the context from which it was called. Even if an error code is returned when an error of some kind is detected, normally no error checking is performed in the context from which the system call is invoked, leaving the possibility that the program will hang. For this reason, these system calls do not return even if an error is detected.

[Items Concerning SMP T-Kernel]

There is no difference from the T-Kernel 1.00 Specification.

tk_ter_tsk: Terminate Task

[C Language Interface]

ER ercd = tk_ter_tsk (ID tskid) ;

[Parameters]

ID	tskid	Task ID	Task ID

[Return Parameters]

ER	ercd	Error Code	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_OBJ	Invalid object state (the target task is in DORMANT state or is the invoking task)
E_DACV	Access Protection Violation

[Description]

Forcibly terminates the task specified in tskid. This system call changes the state of the target task specified in tskid to DORMANT state.

Even if the target task was in a wait state (including SUSPEND state), the wait state is released and the task is terminated. If the target task was in some kind of queue (semaphore wait, etc.), executing tk_ter_tsk results in its removal from the queue.

This system call cannot specify the invoking task. If the invoking task is specified, error code E_OBJ is returned.

Access protection is applied to this system call.

The relationships between target task states and the results of executing tk_ter_tsk are summarized in Table 4.

Table 4: Target Task State and Execution Result (tk_ter_tsk)

Target Task State	tk_ter_tsk ercd Parameter	Processing
RUN or READY state (except for invoking task)	E_OK	Forced termination
RUN state (invoking task)	E_OBJ	No operation
WAIT state (WAITING, SUSPENDED, WAITING-SUSPENDED)	E_OK	Forced termination
DORMANT state	E_OBJ	No operation
NON-EXISTENT state	E_NOEXS	No operation

[Additional Notes]

When a task is terminated by tk_ter_tsk, the resources acquired by the task up to that time (memory blocks, and semaphores, etc.) are not automatically released. The user is responsible for releasing such resources before the task is terminated.

As a rule, the task priority and other information included in the TCB are reset when the task returns to DORMANT state. If, for example, the task priority is changed by tk_chg_pri and later terminated by tk_ter_tsk, the task priority reverts to the startup priority (itskpri) specified when the task was started. It does not keep the task priority in effect at the time tk_ter_tsk was executed.

Forcible termination of another task is intended for use only by a debugger or a few other tasks closely elated to the OS. As a rule, this system call is not to be used by ordinary applications or middleware, for the following reason.

Forced termination occurs irrespective of the running state of the target task. If, for example, a task were forcibly terminated while the task was calling a middleware function, the task would terminate right while the middleware was executing. If such a situation were allowed, normal operation of the middleware could not be guaranteed.

This is an example of how task termination cannot be allowed when the task status (what it is executing) is unknown. Ordinary applications therefore must not use the forcible termination function.

[Items Concerning SMP T-Kernel]
Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.

tk_chg_pri: Change Task Priority

[C Language Interface]

ER ercd = tk_chg_pri (ID tskid, PRI tskpri) ;

[Parameters]

ID	tskid	Task ID	Task ID
PRI	tskpri	Task Priority	Task priority

[Return Parameters]

ER	ercd	Error Code	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_PAR	Parameter error (tskpri is invalid or cannot be used)
E_ILUSE	Illegal use (upper priority limit exceeded)
E_DACV	Access Protection Violation

[Description]

Changes the base priority of the task specified in tskid to the value specified in tskpri. The current priority of the task also changes as a result.

Task priority values are specified from 1 to 140 with the smaller numbers indicating higher priority.

When TSK_SELF (= 0) is specified in tskid, the invoking task is the target task. Note, however, that when tskid = TSK_SELF is specified in a system call issued from a task-independent portion, error code E_ID is returned. When TPRI_INI (= 0) is specified as tskpri, the target task base priority is changed to the initial priority when the task was started (itskpri).

A priority changed by this system call remains valid until the task is terminated. When the task reverts to DORMANT state, the task priority before its exit is discarded, with the task again assigned to the initial priority when the task was started (itskpri). A priority changed while the task is already in DORMANT state, however, becomes valid, so that the task has the new priority as its initial priority the next time it is started.

If as a result of this system call execution the target task current priority matches the base priority (this condition is always met when the mutex function is not used), processing is as follows.

If the target task is in a run state, the task precedence changes according to its priority. The target task has the lowest precedence among tasks of the same priority after the change.

If the target task is in some kind of priority-based queue, the order in that queue changes in accordance with the new task priority. Among tasks of the same priority after the change, the target task is queued at the end.

If the target task has locked a TA_CEILING attribute mutex or is waiting for a lock, and the base priority specified in tskpri is higher than any of the ceiling priorities, error code E_ILUSE is returned.

Access protection is applied to this system call.

[Additional Notes]

In some cases, when this system call results in a change in the queued order of the target task in a task priority-based queue, it may be necessary to release the wait state of another task waiting in that queue (in a message buffer send queue, or in a queue waiting to acquire a variable-size memory pool).

In some cases, when this system call results in a base priority change while the target task is waiting for a TA_INHERIT attribute mutex lock, dynamic priority inheritance processing may be necessary.

When a mutex function is not used and the system call is issued specifying the invoking task as the target task, setting the new priority to the base priority of the invoking task, the order of execution of the invoking task becomes the lowest among tasks of the same priority. This system call can therefore be used to relinquish execution privilege.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.

Change Task Slice-time tk_chg_slt

tk_chg_slt:Change Task Slicetime

[C Language Interface]

ER ercd = tk_chg_slt (ID tskid, RELTIM slicetime) ;

[Parameters]

ID	tskid	TaskID	Task ID
RELTIM	slicetime		Time slice (ms)

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_PAR	Parameter error (slicetime is invalid)
E_DACV	Access Protection Violation

[Description]

Changes the time slice of the task specified in tskid to the value specified in slicetime.

The time slice function is used for round robin scheduling of tasks. When a task runs continuously for the length of time specified in slicetime or longer, its precedence is switched to the lowest among tasks of the same priority, automatically yielding the execution privilege to the next task.

Setting slicetime = 0 indicates unlimited time, and the task does not automatically yield execution privilege. When a task is created, by default it is set to slicetime = 0.

The invoking task can be specified by setting tskid = TSK_SELF = 0. Note, however, that when a system call is issued from a task-independent portion and tskid = TSK_SELF = 0 is specified, error code E_ID is returned.

The time slice as changed by this system call remains valid until the task is terminated. When the task reverts to DORMANT state, the time slice before termination is discarded, and the value at the time of task creation (slicetime = 0) is assigned. A time slice changed while the task is already in DORMANT state, however, becomes valid, being applied the next time the task is started.

Access protection is applied to this system call.

[Additional Notes]

The time while execution privilege is preempted by a higher-priority task does not count in the continuous run time; moreover, even if execution privilege is preempted by a higher-priority task, the run time is not treated as discontinuous. In other words, the time while execution privilege is preempted by a higher-priority task is ignored for the purposes of counting run time.

If the specified task is the only one running at its priority, the time slice is effectively meaningless and the task runs continuously.

If a task of slicetime = 0 is included in tasks of the same priority, as soon as that task obtains execution right, round robin scheduling is stopped.

The method of counting run time is implementation-dependent, but does not need to be especially precise. In fact, applications should not expect very high precision.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

● When the specified task cannot be accessed due to the access protection, E_DACV is returned.

Get Task Space tk_get_tsp

tk_get_tsp: Get Task Space

[C Language Interface]

ER ercd = tk_get_tsp (ID tskid, T_TSKSPC *pk_tskspc) ;

[Parameters]

ID	tskid	Task ID

[Return Parameters]

T_TSKSPC	tskspc	Task space information
ER	ercd	Error code

```
typedef struct t_tskspc {
        VP      uatb;       Task space page table address
        INT     lsid;       Task space ID (logical space ID)
} T_TSKSPC;
```

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_PAR	Parameter error (the return parameter packet address cannot be used)
E_DACV	Access Protection Violation

[Description]

Gets the current task space information for the task specified in tskid.

The invoking task can be specified by setting tskid = TSK_SELF = 0. Note, however, that when a system call is issued from a task-independent portion and tskid = TSK_SELF = 0 is specified, error code E_ID is returned.

Access protection is applied to this system call.

[Additional Notes]

The accuracy of T_TSKSPC (uatb, lsid) is implementation-dependent, but the above definitions should be followed to the extent possible.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.

tk_set_tsp: Set Task Space

[C Language Interface]

```
ER ercd = tk_set_tsp ( ID tskid, T_TSKSPC *pk_tskspc ) ;
```

[Parameters]

ID	tskid	Task ID
T_TSKSPC	tskspc	Task space

```
typedef struct t_tskspc {
    VP      uatb;       Task space page table address
    INT     lsid;       Task space ID (logical space ID)
} T_TSKSPC;
```

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_PAR	Parameter error (pk_tskspc is invalid or cannot be used)
E_DACV	Acess Protection Violation

[Description]

Sets the task space of the task specified in tskid.

The invoking task can be specified by setting tskid = TSK_SELF = 0. Note, however, that when a system call is issued from a task-independent portion and tskid = TSK_SELF = 0 is specified, error code E_ID is returned.

The OS is not aware of the effects of task space changes. If, for example, a task space is changed while a task is using it for its execution, the task may hang or encounter other problems. The caller is responsible for avoiding such problems.

Access protection is applied to this system call.

[Additional Notes]

The accuracy of T_TSKSPC (uatb, lsid) is implementation-dependent, but the above definitions should be followed to the extent possible.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.

Get Task Resource ID tk_get_rid

tk_get_rid: Get Task Resource ID

[C Language Interface]

```
ID resid = tk_get_rid ( ID tskid ) ;
```

[Parameters]

ID	tskid	Task ID

[Return Parameters]

ID	resid	Resource ID or Error Code

[Error Codes]

E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_OBJ	Task does not belong to a resource group
E_DACV	Acess Protection Violation

[Description]

Returns the resource group to which the task specified in tskid currently belongs.

The invoking task can be specified by setting tskid = TSK_SELF = 0. Note, however, that when a system call is issued from a task-independent portion and tskid = TSK_SELF = 0 is specified, error code E_ID is returned.

Access protection is applied to this system call.

[Additional Notes]

If a resource group is deleted, this system call may return the Resource ID of the deleted resource group. Whether or not an error code (E_OBJ) is returned is implementation-dependent. (See tk_cre_res, tk_del_res.)

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.

Set Task Resource ID tk_set_rid

tk_set_rid: Set Task Resource ID

[C Language Interface]

```
ID oldid = tk_set_rid ( ID tskid, ID resid ) ;
```

[Parameters]

ID	tskid	Task ID
ID	resid	New resource ID

[Return Parameters]

ID	oldid	Old resource ID or Error Code

[Error Codes]

E_ID	Invalid ID number (tskid or resid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid or resid does not exist)
E_DACV	Access protection violation

[Description]

Changes the current resource group of the task specified in tskid to the resource group specified in resid. The Resource ID of the old resource group before the change is passed in a return parameter.

The invoking task can be specified by setting tskid = TSK_SELF = 0. Note, however, that when a system call is issued from a task-independent portion and tskid = TSK_SELF = 0 is specified, error code E_ID is returned.

Access protection is applied to this system call.

[Additional Notes]

In some cases error is not returned even if resid was previously deleted. Whether or not an error code (E_NOEXS) is returned is implementation-dependent. In principle it is the responsibility of the caller not to specify a deleted resource group.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.

tk_get_reg: Get Task Registers

[C Language Interface]

ER ercd = tk_get_reg (ID tskid, T_REGS *pk_regs, T_EIT *pk_eit, T_CREGS *pk_cregs) ;

[Parameters]

ID	tskid	Task ID	Task ID

[Return Parameters]

T_REGS	pk_regs	Packet of Registers	General registers
T_EIT	pk_eit	Packet of EIT	Registers saved when EIT occurs
T_CREGS	pk_cregs	Packet of Control Registers	Control registers
ER	ercd	Error Code	Error code

The contents of T_REGS, T_EIT, and T_CREGS are defined for each CPU and implementation.

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_OBJ	Invalid object state (called for the invoking task)
E_CTX	Context error (called from task-independent portion)
E_DACV	Access Protection Violation

[Description]

Gets the current register contents of the task specified in tskid.

If NULL is set in pk_regs, pk_eit, or pk_cregs, the corresponding registers are not referenced.

The referenced register values are not necessarily the values at the time the task portion was executing.

If this system call is issued for the invoking task, error code E_OBJ is returned. Moreover, in SMP T-Kernel, this system call cannot be issued for other tasks in RUN state. In both cases E_OBJ is returned.

Access protection is applied to this system call.

[Additional Notes]

In principle, all registers in the task context can be referenced. This includes not only physical CPU registers but also those treated by the OS as virtual registers.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.
- When the specified task is in RUN state, E_OBJ is returned.

Set Task Registers tk_set_reg

tk_set_reg: Set Task Registers

[C Language Interface]

```
ER ercd = tk_set_reg ( ID tskid, T_REGS *pk_regs, T_EIT *pk_eit, T_CREGS *pk_cregs ) ;
```

[Parameters]

ID	tskid	Task ID	Task ID
T_REGS	pk_regs	Packet of Registers	General registers
T_EIT	pk_eit	Packet of EIT	Registers saved when EIT occurs
T_CREGS	pk_cregs	Packet of Control Registers	Control registers

The contents of T_REGS, T_EIT, and T_CREGS are defined for each CPU and implementation.

[Return Parameters]

ER	ercd	Error Code	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_OBJ	Invalid object state (called for the invoking task or RUN state)
E_CTX	Context error (called from task-independent portion)
E_DACV	Access Protection Violation

[Description]

Sets the current register contents of the task specified in tskid.

If NULL is set in pk_regs, pk_eit, or pk_cregs, the corresponding registers are not set.

The set register values are not necessarily the values while the task portion is executing. The OS is not aware of the effects of register value changes.

If this system call is issued for the invoking task, error code E_OBJ is returned. Moreover, in SMP T-Kernel, this system call cannot be issued for other tasks in RUN state. In both cases E_OBJ is returned.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.
- When the specified task is in RUN state, E_OBJ is returned.

Get Task Co-processor Registers tk_get_cpr

tk_get_cpr:Get Task Coprocessor Registers

[C Language Interface]

ER ercd = tk_get_cpr (ID tskid, INT copno, T_COPREGS *pk_copregs) ;

[Parameters]

ID	tskid	Task ID
INT	copno	Co-processor number (0 to 3)

[Return Parameters]

T_COPREGS	pk_copregs	Co-processor registers
ER	ercd	Error code

```
typedef union {
        T_COP0REG        cop0;    Co-processor number 0 register
        T_COP1REG        cop1;    Co-processor number 1 register
        T_COP2REG        cop2;    Co-processor number 2 register
        T_COP3REG        cop3;    Co-processor number 3 register
} T_COPREG;
```

The contents of T_COPn REG are defined for each CPU and implementation.

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_OBJ	Invalid object state (called for the invoking task or RUN state)
E_CTX	Context error (called from task-independent portion)
E_PAR	Parameter error (copno is invalid or the specified co-processor does not exist)
E_DACV	Access Protection Violation

[Description]

Gets the current contents of the register specified in copno of the task specified in tskid.

The referenced register values are not necessarily the values at the time the task portion was executing.

If this system call is issued for the invoking task, error code E_OBJ is returned. Moreover, in SMP T-Kernel, this system call cannot be issued for other tasks in RUN state. In both cases E_OBJ is returned.

Access protection is applied to this system call.

[Additional Notes]

In principle, all registers in the task context can be referenced. This includes not only physical CPU registers but also those treated by the OS as virtual registers.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.
- When the specified task is in RUN state, E_OBJ is returned.

Set Task Co-processor Registers tk_set_cpr

tk_set_cpr: Set Task Coprocessor Registers

[C Language Interface]

```
ER ercd = tk_set_cpr ( ID tskid, INT copno, T_COPREGS *pk_copregs ) ;
```

[Parameters]

ID	tskid	Task ID
INT	copno	Co-processor number (0 to 3)
T_COPREGS	pk_copregs	Co-processor registers

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_OBJ	Invalid object state (called for the invoking task or RUN state)
E_CTX	Context error (called from task-independent portion)
E_PAR	Parameter error (copno is invalid or the specified co-processor does not exist), or the set register value is invalid (implementation-dependent)
E_DACV	Access Protection Violation

[Description]

Sets the contents of the register specified in copno of the task specified in tskid.

The set register values are not necessarily the values while the task portion is executing. The OS is not aware of the effects of register value changes.

It is possible, however, that some registers or register bits cannot be changed if the OS does not allow such changes (implementation-dependent).

If this system call is issued for the invoking task, error code E_OBJ is returned. Moreover, in SMP T-Kernel, this system call cannot be issued for other tasks in RUN state. In both cases E_OBJ is returned.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.
- When the specified task is in RUN state, E_OBJ is returned.

Get Task Information tk_inf_tsk

tk_inf_tsk: Get Task Information

[C Language Interface]

ER ercd = tk_inf_tsk (ID tskid, T_ITSK *pk_itsk, BOOL clr) ;

[Parameters]

ID	tskid	Task ID
T_ITSK*	pk_itsk	Address of packet for returning task information
BOOL	clr	Clear task information

[Return Parameters]

ER	ercd	Error code

pk_itsk detail:

RELTIM	stime	Cumulative system-level run time (ms)
RELTIM	utime	Cumulative user-level run time (ms)

——(Other implementation-dependent parameters may be added beyond this point.)——

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_PAR	Parameter error (the return parameter packet address cannot be used)
E_DACV	Access Protection Violation

[Description]

Gets statistical information for the task specified in tskid.

If clr = TRUE = 0, the cumulative information is reset (cleared to 0) after getting the information.

The invoking task can be specified by setting tskid = TSK_SELF = 0. Note, however, that when a system call is issued from a task-independent portion and tskid = TSK_SELF = 0 is specified, error code E_ID is returned.

Access protection is applied to this system call.

[Additional Notes]

The system-level run time is that while running at TA_RNG0, and the user-level run time is that while running at protection levels other than TA_RNG0. A task created to run at TA_RNG0 is therefore counted entirely as system-level run time.

The method of counting run time is implementation-dependent, but does not need to be especially precise. In fact, applications should not expect very high precision.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.

tk_ref_tsk: Refer Task Status

[C Language Interface]

```
ER ercd = tk_ref_tsk ( ID tskid, T_RTSK *pk_rtsk ) ;
```

[Parameters]

ID	tskid	TaskID	Task ID
T_RTSK*	pk_rtsk	Packet to Refer Task	Address of packet for returning task status

[Return Parameters]

ER	ercd	ErrorCode	Error code

pk_rtsk detail:

VP	exinf	ExtendedInformation	Extended information
PRI	tskpri	TaskPriority	Current task priority
PRI	tskbpri		Base priority
UINT	tskstat	TaskState	Task state
UINT	tskwait		Wait factor
ID	wid		Waiting object ID
INT	wupcnt		Queued wakeup requests
INT	suscnt		Nested suspend requests
RELTIM	slicetime		Maximum continuous run time allowed (ms)
UINT	waitmask		Disabled wait factors
UINT	texmask		Allowed task exceptions
UINT	tskevent		Task events

——(Other implementation-dependent parameters may be added beyond this point.)——

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_PAR	Parameter error (the return parameter packet address cannot be used)
E_DACV	Access Protection Violation

[Description]

Gets the state of the task specified in tskid.

tskstat takes the following values.

tskstat:

TTS_RUN	0x0001	RUN
TTS_RDY	0x0002	READY
TTS_WAI	0x0004	WAIT
TTS_SUS	0x0008	SUSPEND
TTS_WAS	0x000c	WAITING-SUSPENDED
TTS_DMT	0x0010	DORMANT
TTS_NODISWAI	0x0080	Wait state disabled

Task states such as TTS_RUN and TTS_WAI are expressed by corresponding bits, which is useful when making a complex state decision (e.g., deciding that the state is one of either RUN or READY state). Note that of the above states, TTS_WAS is a combination of TTS_SUS and TTS_WAI, but TTS_SUS is never combined with other states (TTS_RUN, TTS_RDY, and TTS_DMT).

In the case of TTS_WAI (including TTS_WAS), if wait states are disabled by tk_dis_wai, TTS_NODISWAI is set. TTS_NODISWAI is never combined with states other than TTS_WAI.

When tk_ref_tsk is executed for an interrupted task from an interrupt handler, RUN (TTS_RUN) is returned as tskstat.
When tskstat is TTS_WAI (including TTS_WAS), the values of tskwait and wid are as shown in Table 5.

Table 5: Values of tskwait and wid

tskwait	Value	Description	wid
TTW_SLP	0x00000001	Wait caused by tk_slp_tsk	0
TTW_DLY	0x00000002	Wait caused by tk_dly_tsk	0
TTW_SEM	0x00000004	Wait caused by tk_wai_sem	semid
TTW_FLG	0x00000008	Wait caused by tk_wai_flg	flgid
TTW_MBX	0x00000040	Wait caused by tk_rcv_mbx	mbxid
TTW_MTX	0x00000080	Wait caused by tk_loc_mtx	mtxid
TTW_SMBF	0x00000100	Wait caused by tk_snd_mbf	mbfid
TTW_RMBF	0x00000200	Wait caused by tk_rcv_mbf	mbfid
TTW_CAL	0x00000400	Wait caused by tk_rcv_mbf	porid
TTW_ACP	0x00000800	Wait for rendezvous acceptance	porid
TTW_RDV	0x00001000	Wait for rendezvous completion	0
(TTW_CAL|TTW_RDV)	0x00001400	Wait on rendezvous call or wait for rendezvous completion	0
TTW_MPF	0x00002000	Wait for tk_get_mpf	mpfid
TTW_MPL	0x00004000	Wait for tk_get_mpl	mplid
TTW_EV1	0x00010000	Wait for task event #1	0
TTW_EV2	0x00020000	Wait for task event #2	0
TTW_EV3	0x00040000	Wait for task event #3	0
TTW_EV4	0x00080000	Wait for task event #4	0
TTW_EV5	0x00100000	Wait for task event #5	0
TTW_EV6	0x00200000	Wait for task event #6	0
TTW_EV7	0x00400000	Wait for task event #7	0
TTW_EV8	0x00800000	Wait for task event #8	0

When tskstat is not TTS_WAI (including TTS_WAS), both tskwait and wid are 0.
waitmask is the same bit array as tskwait.
For a task in DORMANT state, wupcnt = 0, suscnt = 0, and tskevent = 0.
The invoking task can be specified by setting tskid = TSK_SELF = 0. Note, however, that when a system call is issued from a task-independent portion and tskid = TSK_SELF = 0 is specified, error code E_ID is returned.
When the task specified with tk_ref_tsk does not exist, error code E_NOEXS is returned.
Access protection is applied to this system call.

[Additional Notes]
Even when tskid = TSK_SELF is specified in this system call, the ID of the invoking task is not known. Use tk_get_tid to find out the ID of the invoking task.

[Items Concerning SMP T-Kernel]
Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.

4.2 Task-Dependent Synchronization Functions

Task-dependent synchronization functions achieve synchronization among tasks by direct manipulation of task states. They include functions for task sleep and wakeup, for canceling wakeup requests, for forcibly releasing task WAIT state, for changing a task state to SUSPEND state, and for delaying execution of the invoking task.

Wakeup requests for a task are queued. That is, when it is attempted to wake up a task that is not sleeping, the wakeup request is remembered, and the next time the task is to go to a sleep state (waiting for wakeup), it does not enter that state. Queuing of task wakeup requests is realized by having the task keep a task wakeup request queuing count. When the task is started, this count is cleared to 0.

Suspend requests for a task are nested. That is, if it is attempted to suspend a task already in SUSPEND state (including WAIT-SUSPEND state), the request is remembered and later when it is attempted to resume the task in SUSPEND state (including WAIT-SUSPEND state), it is not resumed. Nesting of suspend requests is realized by having the task keep a suspend request nesting count. When the task is started, this count is cleared to 0.

In SMP T-Kernel, access protection is applied to system calls that specify a task ID.

Service calls with different specifications from the T-Kernel 1.00 Specification are summarized in the table below. For more details refer to the explanation on each system call.

Call Name	Function	Different from T-Kernel 1.00 Specification
tk_slp_tsk	Putting Invoking Task to Sleep	○
tk_wup_tsk	Wakeup Task	△
tk_can_wup	Cancel Wakeup Task	△
tk_rel_wai	Release Wait	△
tk_sus_tsk	Suspend Other Task	△
tk_rsm_tsk	Resume Suspended Task	△
tk_frsm_tsk	Force Resume Task	△
tk_dly_tsk	Delay Task	○
tk_sig_tev	Send Event to Task	△
tk_wai_tev	Wait Task Event	○
tk_dis_wai	Disable Task Wait	△
tk_ena_wai	Enable Task Wait	△

Different from T-Kernel 1.00 Specification ○:No ×:Yes △:Different only in that E_DACV is returned due to the access protection tk_slp_tsk: Sleep Task

[C Language Interface]

```
ER ercd = tk_slp_tsk ( TMO tmout ) ;
```

[Parameters]

TMO	tmout	Timeout	Timeout

[Return Parameters]

ER	ercd	Error Code	Error code

[Error Codes]

E_OK	Normal completion
E_PAR	Parameter error (tmout <= (-2))
E_RLWAI	Wait state released (tk_rel_wai received in wait state)
E_DISWAI	Wait released by wait disabled state
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion or in dispatch disabled state)

[Description]

Changes the state of the invoking task from RUN state to sleep state (WAIT for tk_wup_tsk).

If tk_wup_tsk is issued for the invoking task before the time specified in tmout has elapsed, this system call completes normally. If timeout occurs before tk_wup_tsk is issued, error code E_TMOUT is returned. Specifying tmout = TMO_FEVR =(-1) means waiting forever. In this case, the task stays in waiting state until tk_wup_tsk is issued.

[Additional Notes]

Since tk_slp_tsk is a system call that puts the invoking task into a wait state, tk_slp_tsk can never be nested. It is possible, however, for another task to issue tk_sus_tsk for a task that was put in a wait state by tk_slp_tsk. In this case, the task goes to WAIT-SUSPEND state.

For simply delaying a task, tk_dly_tsk should be used rather than tk_slp_tsk.

The task sleep function is intended for use by applications and as a rule should not be used by middleware.

The reason is that attempting to achieve synchronization by putting a task to sleep in two or more places would cause confusion, leading to mis-operation. For example, if sleep were used by both an application and middleware for synchronization, a wakeup request might arise in the application while middleware has the task sleeping. In such a situation, normal operation would not be possible in either the application or middleware.

Proper task synchronization is not possible because it is not clear where the wait for wakeup originated. Task sleep is often used as a simple means of task synchronization. Applications should be able to use it freely, which means, as a rule, it should not be used by middleware.

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

Wakeup Task tk_wup_tsk

tk_wup_tsk: Wakeup Task

[C Language Interface]

ER ercd = tk_wup_tsk (ID tskid) ;

[Parameters]

ID	tskid	Task ID	Task ID

[Return Parameters]

ER	ercd	Error Code	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_OBJ	Invalid object state (called for the invoking task or for a task in DORMANT state)
E_QOVR	Queuing or nesting overflow (too many queued wakeup requests in wupcnt)
E_DACV	Access Protection Violation

[Description]

If the task specified in tskid was put in WAIT state by tk_slp_tsk, this system call releases the WAIT state.

This system call cannot be called for the invoking task. If the invoking task is specified, error code E_OBJ is returned.

If the target task has not called tk_slp_tsk and is not in WAIT state, the wakeup request by tk_wup_tsk is queued. That is, the calling of tk_wup_tsk for the target task is recorded when tk_slp_tsk is called after that, the task does not go to WAIT state. This is what is meant by queuing of wakeup requests.

The queuing of wakeup requests works as follows. Each task keeps a wakeup request queuing count (wupcnt) in its TCB. Its initial value (when tk_sta_tsk is executed) is 0. When tk_wup_tsk is issued for a task not sleeping (not in WAIT state), the count is incremented by 1; but each time tk_slp_tsk is executed, the count is decremented by 1. When tk_slp_tsk is executed for a task whose wakeup queuing count is 0, the queuing count does not become negative but rather the task goes to WAIT state.

It is always possible to queue tk_wup_tsk one time (wupcnt = 1), but the maximum queuing count (wupcnt) is implementation-dependent and may be set to any appropriate value of 1 or above. In other words, issuing tk_wup_tsk once for a task not in WAIT state does not return error, but whether error is returned for the second or subsequent time tk_wup_tsk is called is an implementation-dependent matter.

When calling tk_wup_tsk causes wupcnt to exceed the maximum allowed value, error code E_QOVR is returned.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.

Cancel Wakeup Task tk_can_wup

tk_can_wup: Cancel Wakeup Task

[C Language Interface]

INT wupcnt = tk_can_wup (ID tskid) ;

[Parameters]

ID	tskid	Task ID	Task ID

[Return Parameters]

INT	wupcnt	Wakeup Count	Number of queued wakeup requests
	or	Error Code	Error Code

[Error Codes]

E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_OBJ	Invalid object state (called for a task in DORMANT state)
E_DACV	Access Protection Violation

[Description]

Passes the wakeup request queuing count (wupcnt) for the task specified in tskid and also cancels all wakeup requests at the same time. That is, this system call clears the wakeup request queuing count (wupcnt) to 0 for the specified task.

The invoking task can be specified by setting tskid = TSK_SELF = 0. Note, however, that when a system call is issued from a task-independent portion and tskid = TSK_SELF = 0 is specified, error code E_ID is returned.

Access protection is applied to this system call.

[Additional Notes]

When processing is performed that involves cyclic wakeup of a task, this system call is used to determine whether the processing was completed within the allotted time. Before processing of a prior wakeup request is completed and tk_slp_tsk is called, the task monitoring this calls tk_can_wup. If wupcnt in the return parameter is 1 or more, it means that the previous wakeup request was not processed within the allotted time. A processing delay or other measure can then be taken accordingly.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.

tk_rel_wai: Release Wait

[C Language Interface]

```
ER ercd = tk_rel_wai ( ID tskid ) ;
```

[Parameters]

ID	tskid	Task ID	Task ID

[Return Parameters]

ER	ercd	Error Code	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_OBJ	Invalid object state (called for a task not in WAIT state (including when called for the invoking task, or for a task in DORMANT state))
E_DACV	Access Protection Violation

[Description]

If the task specified in tskid is in some kind of wait state (not including SUSPEND state), forcibly releases that state.

This system call returns error code E_RLWAI to the task whose WAIT state was released.

Wait release requests by tk_rel_wai are not queued. That is, if the task specified in tskid is already in WAIT state, the WAIT state is cleared; but if it is not in WAIT state when this system call is issued, error code E_OBJ is returned to the caller. Likewise, error code E_OBJ is returned when this system call is issued specifying the invoking task.

The tk_rel_wai system call does not release a SUSPEND state. If it is issued for a task in WAITSUSPEND state, the task goes to SUSPEND state. If it is necessary to release SUSPEND state, the system call tk_frsm_tsk is used instead.

Access protection is applied to this system call.

The states of the target task when tk_rel_wai is called and the results of its execution in each state are shown in Table 6.

Table 6: Task States and Results of tk_rel_wai Execution

Target Task State	tk_rel_tsk ercd Parameter	Processing
Run state (RUN, READY) (not for invoking task)	E_OBJ	No operation
RUN state (for invoking task)	E_OBJ	No operation
WAIT state	E_OK	Wait released *1
(SUSPENDED)	E_OBJ	No operation
(WAITING-SUSPENDED)	E_OK	To SUSPENDED state
DORMANT state	E_OBJ	Wait released.
NON-EXISTENT state	E_NOEXS	Wait released.

* 1 Error code E_RLWAI is returned to the target task. The target task is guaranteed to be released from its wait state without any resource allocation (without the wait release conditions being met).

[Additional Notes]

A function similar to timeout can be realized by using an alarm handler or the like to issue this system call after a given task has been in WAIT state for a set time.

The main differences between tk_rel_wai and tk_wup_tsk are the following.

- Whereas tk_wup_tsk releases only WAIT state effected by tk_slp_tsk, tk_rel_wai also releases WAIT state caused by other factors (tk_wai_flg, tk_wai_sem, tk_rcv_msg, and tk_get_blk, etc.).

- From the respective of the task in WAIT state, release of the WAIT state by tk_wup_tsk returns a Normal completion (E_OK), whereas release by tk_rel_wai returns an error code (E_RLWAI).

- Wakeup requests by tk_wup_tsk are queued if tk_slp_tsk has not yet been executed. If tk_rel_wai is issued for a task not in WAIT state, error code E_OBJ is returned.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.

tk_sus_tsk: Suspend Task

[C Language Interface]

```
ER ercd = tk_sus_tsk ( ID tskid ) ;
```

[Parameters]

ID	tskid	Task ID	Task ID

[Return Parameters]

ER	ercd	Error Code	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_OBJ	Invalid object state (called for the invoking task or for a task in DORMANT state)
E_CTX	A task in RUN state was specified in dispatch disabled state
E_QOVR	Queuing or nesting overflow (too many nested requests in suscnt)
E_DACV	Access Protection Violation

[Description]

Puts the task specified in tskid in SUSPEND state and interrupts execution of the task.

SUSPEND state is released by issuing system call tk_rsm_tsk or tk_frsm_tsk.

If tk_sus_tsk is called for a task already in WAIT state, the state goes to a combination of WAIT state and SUSPEND state (WAIT-SUSPEND state). Thereafter, when the task wait release conditions are met, the task goes to SUSPEND state. If tk_rsm_tsk is issued for a task in WAIT-SUSPEND state, the task state reverts to WAIT state.

Since SUSPEND state means task interruption by a system call issued by another task, this system call cannot be issued for the invoking task. If the invoking task is specified, error code E_OBJ is returned.

When a task in RUN state (RUNNING) is specified in the dispatch disabled state, an E_CTX error results.

If tk_sus_tsk is issued more than once for the same task, the task is put in SUSPEND state multiple times. This is called nesting of suspend requests. In this case, the task reverts to its original state only when tk_rsm_tsk has been issued for the same number of times as tk_sus_tsk (suscnt). Accordingly, nesting of the pair tk_sus_tsk . tk_rsm_tsk is possible.

The limit value of the issue count and whether or not nesting of suspend requests (function to issue tk_sus_tsk for the same task more than once) is supported are implementation- dependent.

If tk_sus_tsk is issued multiple times in a system that does not allow suspend request nesting, or if the nesting count exceeds the allowed limit, error code E_QOVR is returned.

Access protection is applied to this system call.

[Additional Notes]

When a task is in WAIT state for resource acquisition (semaphore wait, etc.) and is also in SUSPEND state, the resource allocation (semaphore allocation, etc.) takes place under the same conditions as when the task is not in SUSPEND state. Resource allocation is not delayed by the SUSPEND state, and there is no change whatsoever in the priority of resource allocation or release from WAIT state. In this way SUSPEND state has an orthogonal relation with other processing and task states.

In order to delay resource allocation to a task in SUSPEND state (temporarily lower its priority), the user can use tk_sus_tsk and tk_rsm_tsk in combination with tk_chg_pri.

Task suspension is intended only for very limited uses closely related to the OS, such as breakpoint processing in a debugger. As a rule it should not be used in ordinary applications or in middleware.

The reason is that task suspension takes place regardless of the running state of the target task. If, for example, a task is put in SUSPEND state while it is calling a middleware function, the task will be stopped in the course of middleware internal processing. In some cases middleware performs resource management or other mutual exclusion control. If a task stops inside middleware while it has resources allocated, other tasks may not be able to use that middleware. This situation can cause a chain reaction, with other tasks stopping and leading to a system-wide deadlock.

For this reason, a task should not be stopped without knowing its status (what it is doing at the time), and ordinary tasks should not use the task suspension function.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.

Resume Task tk_rsm_tsk
Force Resume Task tk_frsm_tsk

tk_rsm_tsk: Resume Task
tk_frsm_tsk: Force Resume Task

[C Language Interface]

```
ER ercd = tk_rsm_tsk ( ID tskid ) ;
ER ercd = tk_frsm_tsk ( ID tskid ) ;
```

[Parameters]

ID	tskid	Task ID	Task ID

[Return Parameters]

ER	ercd	Error Code	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_OBJ	Invalid object state (the specified task is not in SUSPEND state (including when this system call specifies the invoking task or a task in DORMANT state))
E_DACV	Access Protection Violation

[Description]

Releases the SUSPEND state of the task specified in tskid. If the target task was earlier put in SUSPEND state by the tk_sus_tsk system call, this system call releases that SUSPEND state and resumes the task execution.

When the target task is in a combined WAIT state and SUSPEND state (WAIT-SUSPEND state), executing tk_rsm_tsk releases only the SUSPEND state, putting the task in WAIT state.

This system call cannot be issued for the invoking task. If the invoking task is specified, error code E_OBJ is returned.

Executing tk_rsm_tsk once clears only one nested suspend request (suscnt). If tk_sus_tsk was issued more than once for the target task (suscnt>=2), the target task remains in SUSPEND state even after tk_rsm_tsk is executed. When tk_frsm_tsk is issued, on the other hand, all suspend requests are released (suscnt is cleared to 0) even if tk_sus_tsk was issued more than once (suscnt>=2). The SUSPEND state is always cleared, and unless the task was in WAIT-SUSPEND state execution resumes.

When the target task is not Suspend state (SUSPENDED), error code E_OBJ is returned.

Access protection is applied to this system call.

[Additional Notes]

After a task in RUN state or READY state is put in SUSPEND state by tk_sus_tsk and then resumed by tk_rsm_tsk or tk_frsm_tsk, the task has the lowest precedence among tasks of the same priority.

When, for example, the following system calls are executed for tasks A and B of the same priority, the result is as indicated below.

```
tk_sta_tsk (tskid＝task_A, stacd_A);
tk_sta_tsk (tskid＝task_B, stacd_B);
/* By the rule of FCFS, precedence becomes task_A --> task_B. */

tk_sus_tsk (tskid＝task_A);
tk_rsm_tsk (tskid＝task_A);
/* In this case precedence becomes task_B --> task_A. */
```

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.

tk_dly_tsk: Delay Task

[C Language Interface]

```
ER ercd = tk_dly_tsk ( RELTIM dlytim ) ;
```

[Parameters]

RELTIM	dlytim	Delay Time	Delay time

[Return Parameters]

ER	ercd	Error Code	Error code

[Error Codes]

E_OK	Normal completion
E_NOMEM	Insufficient memory
E_PAR	Parameter error (dlytim is invalid)
E_CTX	Context error (issued from task-independent portion or in dispatch disabled state)
E_RLWAI	Wait state released (tk_rel_wai received in wait state)
E_DISWAI	Wait released by wait disabled state

[Description]

Temporarily stops execution of the invoking task and waits for time dlytim to elapse.

The state while the task waits for the delay time to elapse is a WAIT state and is subject to release by tk_rel_wai.

If the task issuing this system call goes to SUSPEND state or WAIT-SUSPEND state while it is waiting for the delay time to elapse, the time continues to be counted in the SUSPEND state.

The time base for dlytim (time unit) is the same as that for system time (= 1 ms).

[Additional Notes]

This system call differs from tk_slp_tsk in that normal completion, not an error code, is returned when the delay time elapses and tk_dly_tsk terminates. Moreover, the wait is not released even if tk_wup_tsk is executed during the delay time. The only way to terminate tk_dly_tsk before the delay time elapses is by calling tk_ter_tsk or tk_rel_wai.

[Items Concerning AMP T-Kernel]

There are no differences with the T-Kernel 1.00 Specification.

tk_sig_tev:Signal Task Event

[C Language Interface]

```
ER ercd = tk_sig_tev ( ID tskid, INT tskevt ) ;
```

[Parameters]

ID	tskid	Task ID
INT	tskevt	tskevt Task event number (1 to 8)

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_OBJ	Invalid object state (called for a task in DORMANT state)
E_PAR	Parameter error (tskevt is invalid)
E_DACV	Access Protection Violation

[Description]

Sends the task event specified in tskevt to the task specified in tskid.

There are eight task event types stored for each task, specified by numbers 1 to 8.

The task event send count is not saved, only whether the event occurs or not.

The invoking task can be specified by setting tskid = TSK_SELF = 0. Note, however, that when a system call is issued from a task-independent portion and tskid = TSK_SELF = 0 is specified, error code E_ID is returned.

Access protection is applied to this system call.

[Additional Notes]

The task event function is used for synchronization much like tk_slp_tsk and tk_wup_tsk, but differs from the use of those system calls in the following ways.

・The wakeup request (task event) count is not kept.

・Wakeup requests can be classified by the eight event types.

Using the same event type for synchronization in two or more places in the same task would cause confusion. Event type allocation should be clearly defined.

The task event function is intended for use in middleware, and as a rule should not be used in ordinary applications. Use of tk_slp_tsk and tk_wup_tsk is recommended for applications.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.

tk_wai_tev: Wait Task Event

[C Language Interface]

INT tevptn = tk_wai_tev (INT waiptn, TMO tmout) ;

[Parameters]

INT	waiptn	Task event pattern
TMO	tmout	Timeout

[Return Parameters]

INT	tevptn	Task event status when wait released or Error Code

[Error Codes]

E_PAR	Parameter error (waiptn or tmout is invalid)
E_RLWAI	Wait state released (tk_rel_wai received in wait state)
E_DISWAI	Wait released by wait disabled state
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion or in dispatch disabled state)

[Description]

Waits for the occurrence of one of the task events specified in waiptn. When the wait is released by a task event, the task events specified in waiptn are cleared (raised task event &= ~waiptn). The task event status occurring when the wait was released (the state before clearing) is passed in the return code (tevptn).

The parameters waiptn and tevptn consist of logical OR values of the bits for each task event in the form 1 << (task event number -1).

A maximum wait time (timeout) can be set in tmout. If the tmout time elapses before the wait release condition is met (tk_sig_tev is not executed), the system call terminates, returning timeout error code E_TMOUT.

Only positive values can be set in tmout. The time base for tmout (time unit) is the same as that for system time (= 1 ms).

When TMO_POL =0 is set in tmout, this means 0 was specified as the timeout value, and E_TMOUT is returned without entering WAIT state even if no task event occurs. When TMO_FEVR = (-1) is set in tmout, this means infinity was specified as the timeout value, and the task continues to wait for a task event without timing out.

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

tk_dis_wai: Disable Task Wait

[C Language Interface]

```
INT tskwait = tk_dis_wai ( ID tskid, UINT waitmask ) ;
```

[Parameters]

ID	tskid	Task ID
UINT	waitmask	Task wait disabled setting

[Return Parameters]

INT	tskwait	Task state after task wait disabled or Error Code

[Error Codes]

E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_PAR	Parameter error (waitmask is invalid)
E_DACV	Access protection violation

[Description]

Disables waits for the wait factors set in waitmask by the task specified in tskid. If the task is already waiting for a factor specified in waitmask, that wait is released.

waitmask is specified as the logical OR of any combination of the following wait factors.

```
#define TTW_SLP     0x00000001  /* Wait caused by sleep */
#define TTW_DLY     0x00000002  /* Wait for task delay */
#define TTW_SEM     0x00000004  /* Wait for semaphore */
#define TTW_FLG     0x00000008  /* Wait for event flag */
#define TTW_MBX     0x00000040  /* Wait for mailbox */
#define TTW_MTX     0x00000080  /* Wait for mutex */
#define TTW_SMBF    0x00000100  /* Wait for message buffer sending */
#define TTW_RMBF    0x00000200  /* Wait for message buffer receipt */
#define TTW_CAL     0x00000400  /* Wait on rendezvous call */
#define TTW_ACP     0x00000800  /* Wait for rendezvous acceptance */
#define TTW_RDV     0x00001000  /* Wait for rendezvous completion */
#define TTW_MPF     0x00002000  /* Wait for fixed-size memory pool */
#define TTW_MPL     0x00004000  /* Wait for variable-size memory pool */
#define TTW_EV1     0x00010000  /* Wait for task event #1 */
#define TTW_EV2     0x00020000  /* Wait for task event #2 */
#define TTW_EV3     0x00040000  /* Wait for task event #3 */
#define TTW_EV4     0x00080000  /* Wait for task event #4 */
#define TTW_EV5     0x00100000  /* Wait for task event #5 */
#define TTW_EV6     0x00200000  /* Wait for task event #6 */
#define TTW_EV7     0x00400000  /* Wait for task event #7 */
#define TTW_EV8     0x00800000  /* Wait for task event #8 */
#define TTX_SVC     0x80000000  /* Extended SVC disabled */
```

TTX_SVC is a special parameter disabling not task wait but the calling of an extended SVC. If TTX_SVC is specified when a task attempts to call an extended SVC, E_DISWAI is returned without calling the extended SVC. This parameter does not have the effect of terminating an already called extended SVC.

In the return code (tskwait), the WAIT state of the task following wait disable processing is returned by tk_dis_wai. This code is the same as tskwait of tk_ref_tsk. Information in regards to TTX_SVC is not returned in tskwait. A tskwait value of 0 means the task has not entered WAIT state (or the wait was released). If tskwait is not 0, this means the task is in WAIT state for a cause other than those disabled in waitmask.

When a task wait is cleared by tk_dis_wai or the task is prevented from entering WAIT state while this system call is in effect, E_DISWAI is returned.

When a system call for which there is the possibility of entering a WAIT state is invoked during wait disabled state, E_DISWAI is returned even if the processing could be performed without waiting. For example, even if message buffer space is available when tk_snd_mbf is called and message buffer sending is possible without entering a WAIT state, E_DISWAI is returned and the message is not sent.

A wait disable set while an extended SVC is executing will be cleared automatically upon return from the extended SVC to its caller. It is automatically cleared also when an extended SVC is called, reverting to the original setting upon return from the extended SVC.

A wait disable setting is cleared also when the task reverts to DORMANT state. The setting made while a task is in DORMANT state, however, is valid and the wait disable is applied the next time the task is started.

In the case of semaphores and most other objects, TA_NODISWAI can be specified when the object is created. An object created with TA_NODISWAI specified cannot have waits disabled, and rejects any wait disable attempt by tk_dis_wai.

The invoking task can be specified by setting tskid = TSK_SELF = 0. Note, however, that when a system call is issued from a task-independent portion and tskid = TSK_SELF = 0 is specified, error code E_ID is returned.

Access protection is applied to this system call.

[Additional Notes]

The wait disable function is provided for preventing execution of an extended SVC handler and is for use mainly (though not exclusively) in break functions.

Wait disable in the case of a rendezvous is more complex than other cases. Essentially, wait disabled state is detected based on a change in the rendezvous wait state, and then the wait is released.

Some specific examples are given here.

When waiting by TTW_CAL is not disabled but TTW_RDV waits are disabled, a task enters into wait on rendezvous call state; but when the rendezvous is accepted and a wait for rendezvous completion would normally begin, the wait is released and E_DISWAI is returned. At this time a message is sent to the receiving task, the receiving task declares acceptance of the message and the task goes to rendezvous established state. Only when the accepting task replies (tk_rpl_rdv) does it become clear that there is no other task in the rendezvous, and error code E_OBJ is returned.

Wait disable applies also when a rendezvous is forwarded. In that case, the attribute of the destination rendezvous port applies. That is, if the TA_NODISWAI attribute is specified for the destination port, wait disable is rejected. If TTW_CAL wait is disabled after going to wait for rendezvous completion state, and a rendezvous is forwarded in that state, the state goes to WAIT on rendezvous call as a result of the forwarding, so wait is disabled by TTW_CAL. In that case, E_DISWAI is returned to both the rendezvous calling task (tk_cal_por) and forwarding task (tk_fwd_por).

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.

Enable task Wait tk_ena_wai

tk_ena_wai: Enable Task Wait

[C Language Interface]

```
ER ercd = tk_ena_wai ( ID tskid ) ;
```

[Parameters]

ID	tskid	Task ID

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_DACV	Access protection violation

[Description]

Releases all wait disable conditions set by tk_dis_wai for the task specified in tskid.

The invoking task can be specified by setting tskid = TSK_SELF = 0. Note, however, that when a system call is issued from a task-independent portion and tskid = TSK_SELF = 0 is specified, error code E_ID is returned.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.

4.3 Task Exception Handling Functions

Task exception handling functions handle exception events occurring in a task, in the context of that task, interrupting normal task processing.

A task exception handler is executed as a part of the task where the task exception occurred, in the context of that task and at the protection level specified when the task was created. The task states in a task exception handler, except for those states concerning task exceptions, are the same as the states when running an ordinary task portion; and the same system calls are available.

A task exception handler can be started only when the target task is running a task portion. If the task is running any other portion when a task exception is raised, the task exception handler is started only after the task returns to the task portion. If a quasi-task portion (extended SVC) is executing when a task exception is raised, a break function corresponding to that extended SVC is called. The break function interrupts the extended SVC processing, and the task returns to the task portion.

Requested task exceptions are cleared when the task exception handler is called (when the task exception handler starts running).

Task exceptions cannot be used for tasks with protection level 0.

Task exceptions are specified by task exception codes from 0 to 31, of which 0 is the highest priority and 31 the lowest. Task exception code 0 is handled differently from the others, as explained below.

Task exception codes 1 to 31:

- These task exception handlers are not nested. A task exception (other than task exception code 0) raised while a task exception handler is running will be made pending.
- On return from a task exception handler, the task resumes from the point where processing was interrupted by the exception.
- It is also possible to use longjmp() or the like to jump to any point in the task without returning from the task exception handler.

Task exception code 0:

- This exception can be nested even while a task exception handler is executing for an exception of task exception code 1 to 31. Execution of task exception code 0 handlers is not nested.
- A task exception handler runs after setting the user stack pointer to the initial setting when the task was started. In a system without a separate user stack and system stack, however, the stack pointer is not reset to its initial setting.
- A task exception code 0 handler does not return to task processing. The task must be terminated.

In SMP T-Kernel, access protection is applied to system calls that specify a Task ID.

Service calls with different specifications from the T-Kernel 1.00 Specification are summarized in the table below. For more details refer to the explanation for each system call.

Call Name	Function	Different from T-Kernel 1.00 Specification
tk_def_tex	Define Task Exception Handler	△
tk_ena_tex	Enable Task Exception	△
tk_dis_tex	Disable Task Exception	△
tk_ras_tex	Raise Task Exception	△
tk_end_tex	End Task Exception Handler	○
tk_ref_tex	Reference Task Exception Status	△

Different from T-Kernel 1.00 Specification ○:No ×:Yes △: Different only in that E_DACV is returned due to the access protection

tk_def_tex: Define Task Exception Handler

[C Language Interface]

ER ercd = tk_def_tex (ID tskid, T_DTEX *pk_dtex) ;

[Parameters]

ID	tskid	Task ID
T_DTEX*	pk_dtex	Task exception handler definition information

pk_dtex detail:

ATR	texatr	Task exception handler attributes
FP	texhdr	Task exception handler address

——(Other implementation-dependent parameters may be added beyond this point.)——

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_NOMEM	Insufficient memory (memory for control block cannot be allocated)
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_OBJ	Invalid object state (the task specified in tskid runs at protection level 0 (TA_RNG0))
E_RSATR	Reserved attribute (texatr, pk_dtex are invalid or cannot be used)
E_PAR	Parameter error (pk_dtex is invalid or cannot be used)
E_DACV	Access Protection Violation

[Description]

Defines a task exception handler for the task specified in tskid. Only one task exception handler can be defined per task; if one is already defined, the last-defined handler is valid. Setting pk_dtex = NULL cancels a definition.

Defining or canceling a task exception handler clears pending task exception requests and disables all task exceptions.

The parameter texatr indicates system attributes in is low bits and implementation-dependent attributes in its high bits. The texatr system attributes are not assigned in the present version, and system attributes are not used.

A task exception handler takes the following form.

```
void texhdr( INT texcd )
  {
      /*
      Task exception handling
      */

      /* Task exception handler termination */
      if ( texcd == 0 ) {
              tk_ext_tsk()or tk_exd_tsk();
      } else {
              tk_end_tex();
              return or longjmp();
      }
  }
```

A task exception handler behaves only like a TA_ASM attribute object and cannot be called via a high-level language support routine. The entry part of the task exception handler must be written in assembly language. The OS vendor must provide the assembly language source of the entry routine for calling the above C language task exception handler. That is, source code equivalent to a high-level language support routine must be provided.

A task set to protection level TA_RNG0 when it is created cannot use task exceptions.

[Additional Notes]
At the time a task is created, no task exception handler is defined and task exceptions are disabled.
When a task reverts to DORMANT state, the task exception handler definition is canceled and task exceptions are disabled. Pending task exceptions are cleared. It is possible, however, to define a task exception handler for a task in DORMANT state.
Task exceptions are software interrupts raised by tk_ras_tex, with no direct relation to CPU exceptions.

[Items Concerning SMP T-Kernel]
Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.

Enable Task Exception tk_ena_tex
Disable Task Exception tk_dis_tex

tk_ena_tex: Enable Task Exception
tk_dis_tex: Disable Task Exception

[C Language Interface]

```
ER ercd = tk_ena_tex ( ID tskid, UINT texptn ) ;

ER ercd = tk_dis_tex ( ID tskid, UINT texptn ) ;
```

[Parameters]

ID	tskid	Task ID
UINT	texmask	Task exception pattern

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist or no task exception handler is defined)
E_PAR	Parameter error (texptn is invalid or cannot be used)
E_DACV	Access Protection Violation

[Description]

Enables or disables task exceptions for the task specified in tskid.

The parameter texptn is a logical OR bit array representing task exception codes in the form 1 << task exception code.

tk_ena_tex enables the task exceptions specified in texptn. tk_dis_tex disables the task exceptions specified in texptn. If the current exception enabled status is texmask, it changes as follows.

Enable: texmask |= texptn

Disable: texmask &= ~texptn

A disabled task exception is ignored, and is not made pending. If exceptions are disabled for a task while there are pending task exceptions, the pending task exception requests are discarded (their pending status is cleared).

Task exceptions cannot be enabled for a task with no task exception handler defined.

These system calls are applicable to tasks in DORMANT state.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.

tk_ras_tex: Raise Task Exception

[C Language Interface]

ER ercd = tk_ras_tex (ID tskid, INT texcd) ;

[Parameters]

ID	tskid	Task ID
INT	texcd	Task exception code (0 to 31)

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist or no task exception handler is defined)
E_OBJ	Invalid object state (called for a task in DORMANT state)
E_PAR	Parameter error (texcd is invalid or cannot be used)
E_CTX	Context error (issued from task-independent portion or in dispatch disabled state)
E_DACV	Access Protection Violation

[Description]

Raises the task exception specified in texcd for the task specified in tskid.

If a task exception handler is already running in the task specified in tskid, the newly raised task exception is made pending. If an exception is pending, a break function is not executed even if the target task is executing an extended SVC.

In the case of texcd = 0, however, exceptions are not made pending even if the target task is executing an exception handler. If the target task is running a task exception handler for an exception of task exception codes 1 to 31, the task exception is accepted; and if an extended SVC is executing, a break function is called. If the target task is running a task exception handler for an exception of task exception code 0, task exceptions are ignored.

The invoking task can be specified by setting tskid = TSK_SELF = 0.

If this system call is issued from a task-independent portion, error code E_CTX is returned.

[Additional Notes]

If the target task is executing an extended SVC, the break handler or the extended SVC runs in the context that called tk_ras_tex. In such a case, tk_ras_tex does not return control until the break function processing ends. Task exceptions raised in the task that called tk_ras_tex while the break function is running are held until the break function ends.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.

tk_end_tex: End Task Exception Handler

[C Language Interface]

```
INT texcd = tk_end_tex ( BOOL enatex ) ;
```

[Parameters]

BOOL	enatex	Task exception handler calling enabled flag

[Return Parameters]

INT	texcd	Raised exception code or Error Code

[Error Codes]

E_CTX Context error (called for other than a task exception handler or task exception code 0 (detection is implementation-dependent))

[Description]

Ends a task exception handler and enables the new task exception handler. If there are pending task exceptions, the highest-priority task exception code among them is passed in the return code. If there are no pending task exceptions, 0 is returned.

If enatex = FALSE and there are pending task exception, calling the new task exception handler is not allowed. In this case, the exception handler specified in return code texcd is in running state upon return from tk_end_tex. If there are no pending task exceptions, calling the new task exception handler is allowed.

If enatex = TRUE, calling the new task exception handler is allowed regardless of whether there are pending task exceptions. Even if there are pending task exceptions, the task exception handler is in terminated status.

There is no way of ending a task exception handler other than by calling tk_end_tex. A task exception handler continues executing from the time it is started until tk_end_tex is called. Even if return is made from a task exception handler without calling tk_end_tex, the task exception handler will still be running at the point of return. Similarly, even if longjmp is used to get out of a task exception handler without calling tk_end_tex, the task exception handler will still be running at the jump destination.

Calling tk_end_tex while task exceptions are pending results in a new task exception being accepted. At this time even when tk_end_tex is called from an extended SVC handler, a break function cannot be called for that extended SVC handler. If extended SVC calls are nested, then when the extended SVC nesting goes down one level, the break function corresponding to the extended SVC return destination can be called. Calling of a task exception handler takes place upon return to the task portion.

The tk_end_tex system call cannot be issued in the case of task exception code 0, since the task exception handler cannot be ended. The behavior when tk_end_tex is called for a handler of task exception code 0 is undefined (implementation-dependent).

This system call cannot be issued from other than a task exception handler. The behavior when it is called from other than a task exception handler is undefined (implementation-dependent).

[Additional Notes]

When tk_end_tex (TRUE) is set and there are pending task exceptions, another task exception handler call is made immediately following tk_end_tex. Moreover, for that reason a task exception handler is called without restoring the stack, giving rise to possible stack overflow.

Ordinarily tk_end_tex (FALSE) can be used, and processing looped as illustrated below while there are task exceptions pending.

```
void texhdr( INT texcd )
{
        if ( texcd == 0 ) tk_exd_tsk();
        do {
              /*
```

```
                                        Task exception handling
                        */
        } while ( (texcd = tk_end_tex(FALSE)) > 0 );
}
```

Strictly speaking, if a task exception were to occur during the interval after 0 is returned by tk_ena_tex ending the loop and before exit from texhdr, the possibility exists of reentering texhdr without restoring the stack. Since task exceptions are software driven, however, ordinarily they do not occur with no relation to tasks; so in practice this is not a problem.

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

Reference task Exception Status tk_ref_tex

tk_ref_tex: Refer Task Exception Status

[C Language Interface]

ER ercd = tk_ref_tex (ID tskid, T_RTEX *pk_rtex) ;

[Parameters]

ID	tskid	Task ID
T_RTEX*	pk_rtex	Address of packet for returning task exception status

[Return Parameters]

ER	ercd	Error code

pk_rtex detail:

UINT	pendtex	Pending task exceptions
UINT	texmask	Allowed task exceptions

──(Other implementation-dependent parameters may be added beyond this point.)──

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_PAR	Parameter error (the return parameter packet address cannot be used)
E_DACV	Access Protection Violation

[Description]

Gets the status of task exceptions for the task specified in tskid.

pendtex indicates the currently pending task exceptions. A raised task exception is indicated in pendtex from the time the task exception is raised until its task exception handler is called.

texmask indicates allowed task exceptions.

Both pendtex and texmask are bit arrays of the form 1 << task exception code.

The invoking task can be specified by setting tskid = TSK_SELF = 0. Note, however, that when a system call is issued from a task-independent portion and tskid = TSK_SELF = 0 is specified, error code E_ID is returned.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified task cannot be accessed due to the access protection, E_DACV is returned.

4.4 Synchronization and Communication Functions

Synchronization and communication functions use objects independent of tasks to synchronize tasks and achieve communication between tasks. The objects available for these purposes include semaphores, event flags and mailboxes.

4.4.1 Semaphore

A semaphore is an object indicating the availability of a resource and its quantity as a numerical value. A semaphore is used to realize mutual exclusion control and synchronization when using a resource. Functions are provided for creating and deleting a semaphore, acquiring and returning resources corresponding to semaphores, and referencing semaphore status. A semaphore is an object identified by an ID number called a semaphore ID.

A semaphore contains a resource count indicating whether the corresponding resource exists and in what quantity, and a queue of tasks waiting to acquire the resource. When a task (the task making event notification) returns m resources, it increments the semaphore resource count by m. When a task (the task waiting for an event) acquires n resources, it decreases the semaphore resource count by n. If the number of semaphore resources is insufficient (i.e., further reducing the semaphore resource count would cause it to become a negative value), a task attempting to acquire resources goes into WAIT state until the next time resources are returned. A task waiting for semaphore resources is put in the semaphore queue.

To prevent too many resources from being returned to a semaphore, a maximum resource count can be set for each semaphore. An error is reported if it is attempted to return resources to a semaphore that would cause this maximum count to be exceeded.

In SMP T-Kernel, access protection is applied to system calls that specify a semaphore ID.

Service calls with different specifications from the T-Kernel 1.00 Specification are summarized in the table below. For more details refer to the explanation for each system call.

Call Name	Function	Different from T-Kernel 1.00 Specification
tk_cre_sem	Create Semaphore	×
tk_del_sem	Delete Semaphore	△
tk_sig_sem	Signal Semaphore	△
tk_wai_sem	Wait for Semaphore Resource	△
tk_ref_sem	Reference Semaphore Status	△

Different from T-Kernel 1.00 Specification ○:No ×:Yes △:Different only in that E_DACV is returned due to the access protection tk_cre_sem: Create Semaphore

[C Language Interface]

```
ID semid = tk_cre_sem ( T_CSEM *pk_csem ) ;
```

[Parameters]

T_CSEM* pk_csem Packet to Create Semaphore   Information about the semaphore to be created

pk_csem detail:

VP	exinf	ExtendedInformation	Extended information
ATR	sematr	SemaphoreAttribute	Semaphore attributes
INT	isemcnt	InitialSemaphoreCount	Initial semaphore count
INT	maxsem	MaximumSemaphoreCount	Maximum semaphore count
ID	domid	DomainID	Domain ID
UB	oname[8]	Object name	DS Object name

—(Other implementation-dependent parameters may be added beyond this point.)—

[Return Parameters]

ID	semid	SemaphoreID	Semaphore ID
	or	ErrorCode	Error Code

[Error Codes]

E_NOMEM	Insufficient memory (memory for control block cannot be allocated)
E_LIMIT	Semaphore count exceeds the system limit
E_RSATR	Reserved attribute (sematr is invalid or cannot be used)
E_PAR	Parameter error (pk_csem is invalid; isemcnt or maxsem is negative or invalid)
E_ID	Invalid ID number (domid is invalid or cannot be used)
E_NOEXS	Object does not exist (domain of domid does not exist)
E_ONAME	Specified object name has already been used

[Description]

Creates a semaphore, assigning it to a semaphore ID. This system call allocates a control block to the created semaphore, setting the initial count to isemcnt and maximum count (upper limit) to maxsem. It must be possible to set maxsem to at least 65535. Whether values above 65536 can be set is implementation-dependent.

exinf can be used freely by the user to store miscellaneous information about the created semaphore. The information set in this parameter can be referenced by tk_ref_sem. If a larger area is needed for indicating user information, or if the information needs to be changed after the semaphore is created, this can be done by allocating separate memory for this purpose and putting the memory packet address in exinf. The OS pays no attention to the contents of exinf.

sematr indicates system attributes in its low bits and implementation-dependent information in the high bits. The system attributes part of sematr is as follows.

sematr:= (TA_TFIFO || TA_TPRI) | (TA_FIRST || TA_CNT) | [TA_ONAME] | [TA_NODISWAI]
| [TA_DOMID] | [(TA_PROTECTED || TA_PRIVATE || TA_PUBLIC)]

TA_TFIFO	Tasks are queued in FIFO order
TA_TPRI	Tasks are queued in priority order
TA_FIRST	The first task in the queue has precedence
TA_CNT	Tasks with fewer requests have precedence
TA_ONAME	Specifies DS Object name
TA_NODISWAI	Wait disabling by tk_dis_wai is prohibited
TA_DOMID	Specifies the domain to which the task belongs
TA_PROTECTED	Sets the access protection attribute to protect
TA_PRIVATE	Sets the access protection attribute to private

TA_PUBLIC Sets the access protection attribute to public

The queuing order of tasks waiting for a semaphore can be specified in TA_TFIFO or TA_TPRI. If the attribute is TA_TFIFO, tasks are ordered by FIFO, whereas TA_TPRI specifies queuing of tasks in order of their priority setting.

TA_FIRST and TA_CNT specify precedence of resource acquisition. TA_FIRST and TA_CNT do not change the order of the queue, which is determined by TA_TFIFO or TA_TPRI.

When TA_FIRST is specified, resources are allocated starting from the first task in the queue regardless of request count. As long as the first task in the queue cannot obtain the requested number of resources, tasks behind it in the queue are prevented from obtaining resources.

TA_CNT means resources are assigned based on the order in which tasks are able to obtain the requested number of resources. The request counts are checked starting from the first task in the queue, and tasks to which their requested amounts can be allocated receive the resources. This is not the same as allocating in order of fewest requests.

When TA_ONAME is specified, oname is valid and is set as the object name. When TA_ONAME is not specified, the object name is not set. The object name must be unique within the domain to which the semaphore belongs. When an object name that has already been used with another semaphore is specified, E_ONAME is returned. When the length of the character string specified for oname is 0 (initial character is termination 0), the object name is not set regardless of the specification of TA_ONAME.

When TA_DOMID is specified, domid is valid, and the domain of domid is set as the domain to which it belongs. When TA_DOMID is not specified, domid is ignored and is the domain to which the kernel domain belongs.

TA_PROTECTED, TA_PRIVATE, and TA_PUBLIC specify the access protection attribute of the semaphore. When either of the access protection attributes is not specified, the access protection is set to the public attribute. In the combination of the domain to which the task belongs and the access protection attribute, semaphores that invoking tasks cannot access due to access protection cannot be created. When the corresponding specification is done, E_PAR is returned.

```
#define TA_TFIFO        0x00000000   /* manage queue by FIFO */
#define TA_TPRI         0x00000001   /* manage queue by priority */
#define TA_FIRST        0x00000000   /* first task in queue has precedence */
#define TA_CNT          0x00000002   /* tasks with fewer requests have precedence */
#define TA_ONAME        0x00000040   /* DS Object name is specified */
#define TA_NODISWAI     0x00000080   /* reject wait disabling */
#define TA_DOMID        0x00010000   /* specify the domain */
#define TA_PRIVATE      0x00040000   /* set the protection attribute to private */
#define TA_PROTECTED    0x00080000   /* set the protection attribute to protect*/
#define TA_PUBLIC       0x00000000   /* set the protection attribute to public */
```

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- TA_DOMID, TA_PROTECTED, TA_PRIVATE, and TA_PUBLIC were added to the semaphore attribute, and the domain to which it belongs and the access protection attribute are specifiable.
- The DS Object name was abolished, and replaced by the establishment of the object name. While the former was a name for debugging, the latter is a name which can be used in general for searching domain ID's, etc. The object name cannot use the same name with the same type of object in the same domain.

Delete Semaphore tk_del_sem

tk_del_sem:Delete Semaphore

[C Language Interface]

ER ercd = tk_del_sem (ID semid) ;

[Parameters]

ID	semid	SemaphoreID	Semaphore ID

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (semid is invalid or cannot be used)
E_NOEXS	Object does not exist (the semaphore specified in semid does not exist)
E_DACV	Access Protection Violation

[Description]

Deletes the semaphore specified in semid.

The semaphore ID and control block area are released as a result of this system call.

This system call completes normally even if there is a task waiting on the semaphore, but error code E_DLT is returned to the task in WAIT state.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified semaphore cannot be accessed due to access protection, E_DACV is returned.

Signal Semaphore tk_sig_sem

tk_sig_sem: Signal Semaphore

[C Language Interface]

ER ercd = tk_sig_sem (ID semid, INT cnt) ;

[Parameters]

ID	semid	SemaphoreID	Semaphore ID
INT	cnt		Resource return count

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (semid is invalid or cannot be used)
E_NOEXS	Object does not exist (the semaphore specified in semid does not exist)
E_QOVR	Queuing or nesting overflow (semcnt is higher than the limit)
E_PAR	Parameter error (cnt <= 0)
E_DACV	Access Protection Violation

[Description]

Returns to the semaphore specified in semid the number of resources indicated in cnt. If there is a task waiting for the semaphore, its request count is checked and resources allocated if possible. A task that has been allocated resources goes to READY state. In some conditions, more than one task may be allocated resources and put in the READY state.

If the semaphore count increases to the point where the maximum count (maxcnt) would be exceeded by the return of more resources, error code E_QOVR is returned. In this case, no resources are returned and the count (semcnt) does not change.

Access protection is applied to this system call.

[Additional Notes]

An error is not returned even if semcnt exceeds the semaphore initial count (isemcnt). When semaphores are used for synchronization (in place of tk_wup_tsk and tk_slp_tsk) and not for mutual exclusion control, the semaphore count (semcnt) will sometimes exceed the initial setting (isemcnt). The semaphore function can be used for mutual exclusion control by setting isemcnt and the maximum semaphore count (maxsem) to the same value and checking for failure of mutual exclusion by catching E_QVR.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified semaphore cannot be accessed due to access protection, E_DACV is returned.

tk_wai_sem:Wait on Semaphore

[C Language Interface]

ER ercd = tk_wai_sem (ID semid, INT cnt, TMO tmout) ;

[Parameters]

ID	semid	SemaphoreID	Semaphore ID
INT	cnt		Resource request count
TMO	tmout	Timeout	imeout

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (semid is invalid or cannot be used)
E_NOEXS	Object does not exist (the semaphore specified in semid does not exist)
E_PAR	Parameter error (tmout <= (-2), cnt <= 0)
E_DLT	The object being waited for was deleted (the specified semaphore was deleted while waiting)
E_RLWAI	Wait state released (tk_rel_wai received in wait state)
E_DISWAI	Wait released by wait disabled state
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion or in dispatch disabled state)
E_DACV	Access Protection Violation

[Description]

From the semaphore specified in semid, the number of resources indicated in cnt. If the requested resources can be allocated, the task issuing this system call does not enter WAIT state but continues executing. In this case, the semaphore count (semcnt) is decreased by the value of cnt. If the resources are not available, the task issuing this system call enters WAIT state, and is put in the queue of tasks waiting for the semaphore. The semaphore count (semcnt) for this semaphore does not change in this case.

A maximum wait time (timeout) can be set in tmout. If the tmout time elapses before the wait release condition is met (tk_sig_sem is not executed), the system call terminates returning timeout error code E_TMOUT.

Only positive values can be set in tmout. The time base for tmout (time unit) is the same as that for system time (= 1 ms).

When TMO_POL = 0 is set in tmout, this means 0 was specified as the timeout value, and E_TMOUT is returned without entering WAIT state even if no resources are acquired. When TMO_FEVR = (-1) is set in tmout, this means infinity was specified as the timeout value and the task continues to wait for resource acquisition without timing out.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified semaphore cannot be accessed due to access protection, E_DACV is returned.

Reference Semaphore Status tk_ref_sem

tk_ref_sem:Refer Semaphore Status

[C Language Interface]

```
ER ercd = tk_ref_sem ( ID semid, T_RSEM *pk_rsem ) ;
```

[Parameters]

ID	semid	SemaphoreID	Semaphore ID
T_RSEM*	pk_rsem	Packet to Refer Semaphore	Address of packet for returning status information

[Return Parameters]

ER	ercd	ErrorCode	Error code

pk_rsem detail:

VP	exinf	ExtendedInformation	Extended information
ID	wtsk	WaitTaskInformation	Waiting task information
INT	semcnt	SemaphoreCount	Semaphore count

──(Other implementation-dependent parameters may be added beyond this point.)──

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (semid is invalid or cannot be used)
E_NOEXS	Object does not exist (the semaphore specified in semid does not exist)
E_PAR	Parameter error (address of the return parameter packet cannot be used)
E_DACV	Access Protection Violation

[Description]

References the status of the semaphore specified in semid, passing in the return parameters the current semaphore count (semcnt), information on tasks waiting for the semaphore (wtsk), and extended information (exinf).

wtsk indicates the ID of a task waiting for the semaphore. If there are two or more such tasks, the ID of the task at the head of the queue is returned. If there are no waiting tasks, wtsk = 0 is returned.

If the specified semaphore does not exist, error code E_NOEXS is returned.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified semaphore cannot be accessed due to access protection, E_DACV is returned.

4.4.2 Event Flag

An event flag is an object used for synchronization, consisting of a pattern of bits used as flags to indicate the existence of the corresponding event. Functions are provided for creating and deleting an event flag, for event flag setting and clearing, event flag waiting, and referring event flag status. An event flag is identified by an ID number, called an event flag ID.

In addition to the bit pattern indicating the existence of corresponding events, an event flag has a queue of tasks waiting for the event flag. The event flag bit pattern is sometimes simply called event flag. The event notifier sets or clears the specified bits of the event flag. A task can be made to wait for all or some of the event flag bits to be set. A task waiting for an event flag is put in the queue of that event flag.

In SMP T-Kernel, access protection is applied to system calls that specify an event flag ID.

Service calls with different specifications from the T-Kernel 1.00 Specification are summarized in the table below. For more details refer to the explanation for each system call.

Call Name	Function	Different from T-Kernel 1.00 Specification
tk_cre_flg	Create Event Flag	×
tk_del_flg	Delete Event Flag	△
tk_set_flg	Set Event Flag	△
tk_clr_flg	Clear Event Flag	△
tk_wai_flg	Wait Event Flag	△
tk_ref_flg	Reference Event Flag Status	△

Different from T-Kernel 1.00 Specification ○:No ×:Yes △: Different only in that E_DACV is returned due to the access protection

tk_cre_flg: Create EventFlag

[C Language Interface]

ID flgid = tk_cre_flg (T_CFLG *pk_cflg) ;

[Parameters]

T_CFLG* pk_cflg	Packet to Create EventFlag	Information about the event flag to be created

pk_cflg detail:

VP	exinf	ExtendedInformation	Extended information
ATR	flgatr	EventFlagAttribute	Event flag attributes
UINT	iflgptn	InitialEventFlagPattern	Initial event flag pattern
ID	domid	DomainID	Domain ID
UB	oname[8]	Object name	DS Object name

——(Other implementation-dependent parameters may be added beyond this point.)——

[Return Parameters]

ID	flgid	EventFlagID	Event flag ID
	or	ErrorCode	Error Code

[Error Codes]

E_NOMEM	Insufficient memory (memory for control block cannot be allocated)
E_LIMIT	Number of event flags exceeds the system limit
E_RSATR	Reserved attribute (flgatr is invalid or cannot be used)
E_PAR	Parameter error (pk_cflg is invalid)
E_ID	Invalid ID number (domid is invalid or cannot be used)
E_NOEXS	Object does not exist (domain of domid does not exist)
E_ONAME	Specified object name has already been used

[Description]

Creates an event flag, assigning it to an event flag ID. This system call allocates a control block to the created event flag and sets its initial value to iflgptn.

In T-Kernel, one word's worth of bits of the processor is grouped with one event flag. Operation is all in units of one word.

exinf can be used freely by the user to store miscellaneous information about the created event flag. The information set in this parameter can be referenced by tk_ref_flg. If a larger area is needed for indicating user information, or if the information needs to be changed after the event flag is created, this can be done by allocating separate memory for this purpose and putting the memory packet address in exinf. The OS pays no attention to the contents of exinf.

flgatr indicates system attributes in its low bits and implementation-dependent information in the high bits. The system attributes part of flgatr is as follows.

flgatr:= (TA_TFIFO || TA_TPRI) | (TA_WMUL || TA_WSGL) | [TA_ONAME] | [TA_NODISWAI]
    | [TA_DOMID] | [(TA_PROTECTED || TA_PRIVATE || TA_PUBLIC)]

TA_TFIFO	Tasks are queued in FIFO order
TA_TPRI	Tasks are queued in priority order
TA_WSGL	Waiting for multiple tasks is not allowed (Wait Single Task)
TA_WMUL	Waiting for multiple tasks is allowed (Wait Multiple Task)
TA_ONAME	Specifies DS Object name

TA_NODISWAI	Wait disabling by tk_dis_wai is prohibited
TA_DOMID	Specifies the domain to which the task belongs
TA_PROTECTED	Sets the access protection attribute to protect
TA_PRIVATE	Sets the access protection attribute to private
TA_PUBLIC	Sets the access protection attribute to public

When TA_WSGL is specified, multiple tasks cannot be in WAIT state at the same time. Specifying TA_WMUL allows waiting by multiple tasks at the same time.

The queuing order of tasks waiting for an event flag can be specified in TA_TFIFO or TA_TPRI. If the attribute is TA_TFIFO, tasks are ordered by FIFO, whereas TA_TPRI specifies queuing of tasks in order of their priority setting. When TA_WSGL is specified, however, since tasks cannot be queued, TA_TFIFO or TA_TPRI makes no difference.

When multiple tasks are waiting for an event flag, tasks are checked in order from the head of the queue, and the wait is released for tasks meeting the conditions. The first task to have its WAIT state released is therefore not necessarily the first in the queue. If multiple tasks meet the conditions, wait state is released for each of them.

When TA_ONAME is specified, oname is valid and is set as the object name. When TA_ONAME is not specified, the object name is not set. The object name must be unique within the domain to which the event flag belongs. When an object name that has already been used with another event flag is specified, E_ONAME is returned. When the length of the character string specified for oname is 0 (initial character is termination 0), the object name is not set regardless of the specification of TA_ONAME.

When TA_DOMID is specified, domid is valid, and the domain of domid is set as the domain to which it belongs.

When TA_DOMID is not specified, domid is ignored and is the domain to which the kernel domain belongs. TA_PROTECTED, TA_PRIVATE, and TA_PUBLIC specify the access protection attribute of the event flag. When either of the access protection attributes is not specified, the access protection is set to the public attribute. In the combination of the domain to which the task belongs and the access protection attribute, event flags that invoking tasks cannot access due to access protection cannot be created. When the corresponding specification is done, E_PAR is returned.

```
#define TA_TFIFO        0x00000000    /* manage queue by FIFO */
#define TA_TPRI         0x00000001    /* manage queue by priority */
#define TA_WSGL         0x00000000    /* prohibit multiple task waiting */
#define TA_WMUL         0x00000008    /* allow multiple task waiting */
#define TA_ONAME        0x00000040    /* DS Object name */
#define TA_NODISWAI     0x00000080    /* prohibit wait disabling */
#define TA_DOMID        0x00010000    /* specify the domain */
#define TA_PRIVATE      0x00040000    /* set the protection attribute to private */
#define TA_PROTECTED    0x00080000    /* set the protection attribute to protect */
#define TA_PUBLIC       0x00000000    /* set the protection attribute to public */
```

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- TA_DOMID, TA_PROTECTED, TA_PRIVATE, and TA_PUBLIC were added to the event flag attribute, and the domain to which it belongs and the access protection attribute are specifiable.
- The DS Object name was abolished, and replaced by the establishment of the object name. While the former was a name for debugging, the latter is a name which can be used in general for searching domain ID's, etc. The object name cannot use the same name with the same type of object in the same domain.

Delete Event Flag tk_del_flg

tk_del_flg:Delete EventFlag

[C Language Interface]

```
ER ercd = tk_del_flg ( ID flgid ) ;
```

[Parameters]

ID	flgid	EventFlagID	Event flag ID

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (flgid is invalid or cannot be used)
E_NOEXS	Object does not exist (the event flag specified in flgid does not exist)
E_DACV	Access Protection Violation

[Description]

Deletes the event flag specified in flgid.

Issuing this system call releases the corresponding event flag ID and control block memory space.

This system call is completed normally even if there are tasks waiting for the event flag, but error code E_DLT is returned to each task waiting on this event flag.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified event flag cannot be accessed due to access protection, E_DACV is returned.

Set Event Flag tk_set_flg
Clear Event Flag tk_clr_flg

tk_set_flg:Set EventFlag
tk_clr_flg: Clear EventFlag

[C Language Interface]

```
ER ercd = tk_set_flg ( ID flgid, UINT setptn ) ;
ER ercd = tk_clr_flg ( ID flgid, UINT clrptn ) ;
```

[Parameters (tk_set_flg)]

ID	flgid	EventFlagID	Event flag ID
UINT	setptn	SetBitPattern	Bit pattern to be set

[Parameters (tk_clr_flg)]

ID	flgid	EventFlagID	Event flag ID
UINT	clrptn	ClearBitPattern	Bit pattern to be cleared

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (flgid is invalid or cannot be used)
E_NOEXS	Object does not exist (the event flag specified in flgid does not exist)
E_DACV	Access Protection Violation

[Description]

tk_set_flg sets the bits indicated in setptn in the one-word event flag specified in flgid. i.e., a logical sum is taken of the values of the event flag specified in flgid and the values indicated in setptn. tk_clr_flg clears the bits of the one-word event flag based on the corresponding zero bits of clrptn. i.e., a logical product is taken of the values of the event flag specified in flgid and the values indicated in clrptn.
After event flag values are changed by tk_set_flg, if the condition for releasing the wait state of a task that called tk_wai_flg is met, the WAIT state of that task is cleared, putting it in RUN state or READY state (or SUSPEND state if the waiting task was in WAIT-SUSPEND state).

Issuing tk_clr_flg never results in wait conditions being released for a task waiting for the specified event flag; thus, dispatching never occurs as a result of calling tk_clr_flg.

Nothing will happen to the event flag if all bits of setptn are cleared to 0 with tk_set_flg or if all bits of clrptn are set to 1 with tk_clr_flg. No error will result in either case.

Multiple tasks can wait for a single event flag if that event flag has the TA_WMUL attribute. The event flag in that case has a queue for the waiting tasks. A single tk_set_flg call for such an event flag may result in the release of multiple waiting tasks.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified event flag cannot be accessed due to access protection, E_DACV is returned.

tk_wai_flg: Wait EventFlag

[C Language Interface]

ER ercd = tk_wai_flg (ID flgid, UINT waiptn, UINT wfmode, UINT *p_flgptn, TMO tmout) ;

[Parameters]

ID	flgid	EventFlagID	Event flag ID
UINT	waiptn	WaitBitPattern	Wait bit pattern
UINT	wfmode	WaitEventFlagMode	Wait release condition
TMO	tmout	Timeout	timeout

[Return Parameters]

ER	ercd	ErrorCode	Error code
UINT	flgptn	EventFlagBitPattern	Event flag bit pattern

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (flgid is invalid or cannot be used)
E_NOEXS	Object does not exist (the event flag specified in flgid does not exist)
E_PAR	Parameter error (waiptn = 0, wfmode is invalid, or tmout <= (-2))
E_OBJ	Invalid object state (multiple tasks are waiting for an event flag with TA_WSGL attribute)
E_DLT	The object being waited for was deleted (the specified event flag was deleted while waiting)
E_RLWAI	Wait state released (tk_rel_wai received in wait state)
E_DISWAI	Wait released by wait disabled state
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion or in dispatch disabled state)
E_DACV	Access Protection Violation

[Description]

Waits for the event flag specified in flgid to be set, fulfilling the wait release condition specified in wfmode.

If the event flag specified in flgid already meets the wait release condition set in wfmode, the waiting task continues executing without going to WAIT state.

wfmode is specified as follows.

wfmode := (TWF_ANDW || TWF_ORW) | [TWF_CLR || TWF_BITCLR]

TWF_ANDW	0x00	AND wait condition
TWF_ORW	0x01	OR wait condition
TWF_CLR	0x10	Clear all
TWF_BITCLR	0x20	Clear condition bit only

If TWF_ORW is specified, the issuing task waits for any of the bits specified in waiptn to be set for the event flag specified in flgid (OR wait). If TWF_ANDW is specified, the issuing task will wait for all of the bits specified in waiptn to be set for the event flag specified in flgid (AND wait).

If TWF_CLR specification is not specified, the event flag values will remain unchanged even after the conditions have been satisfied and the task has been released from WAIT state. If TWF_CLR is specified, all bits of the event flag will be cleared to 0 once wait conditions of the waiting task have been met. If TWF_BITCLR is specified, then when the conditions are met and the task is released from WAIT state, only the bits matching the event flag wait release conditions are cleared to 0 (event flag values &= ~wait release conditions).

The return parameter flgptn returns the value of the event flag after the WAIT state of a task has been released due to this system call. If TWF_CLR or TWF_BITCLR was specified, the value before event flag bits were cleared is returned. The value returned by flgptn fulfills the wait release conditions of this system call. The contents of flgptn are indeterminate if the wait is released due to other reasons such as timeout.

A maximum wait time (timeout) can be set in tmout. If the tmout time elapses before the wait release condition is met, the system call terminates, returning timeout error code E_TMOUT.

Only positive values can be set in tmout. The time base for tmout (time unit) is the same as that for system time (= 1 ms).

When TMO_POL = 0 is set in tmout, this means 0 was specified as the timeout value, and E_TMOUT is returned without entering WAIT state even if the condition is not met. When TMO_FEVR = (-1) is set in tmout, this means infinity was specified as the timeout value, and the task continues to wait for the condition to be met without timing out.

In the case of a timeout, the event flag bits are not cleared even if TWF_CLR or TWF_BITCLR was specified.

Setting waiptn to 0 results in Parameter error E_PAR.

A task cannot execute tk_wai_flg for an event flag having the TA_WSGL attribute while another task is waiting for it. Error code E_OBJ will be returned for the task issuing the subsequent tk_wai_flg, regardless of whether that task would have gone to WAIT state; i.e., regardless of whether the wait release conditions would be met.

If an event flag has the TA_WMUL attribute, multiple tasks can wait for it at the same time. Waiting tasks can be queued, and the WAIT states of multiple tasks can be released by issuing tk_set_flg just once.

If multiple tasks are queued for an event flag with TA_WMUL attribute, the behavior is as follows.

- Tasks are queued in either FIFO or priority order. (Release of wait state does not always start from the head of the queue, however, depending on factors such as waiptn and wfmode settings.)

- If TWF_CLR or TWF_BITCLR was specified by a task in the queue, the event flag is cleared when that task is released from WAIT state.
- Tasks later in the queue than a task specifying TWF_CLR or TWF_BITCLR will see the event flag after it has already been cleared.

If multiple tasks having the same priority are simultaneously released from waiting as a result of tk_set_flg, the order of tasks in the ready queue (precedence) after release will continue to be the same as their original order in the event flag queue.

Access protection is applied to this system call.

[Additional Notes]

If a logical sum of all bits is specified as the wait release condition when tk_wai_flg is called (waiptn=0xfff... ff, wfmode=TWF_ORW), it is possible to transfer messages using one-word bit patterns in combination with tk_set_flg. However, it is not possible to send a message containing only 0s for all bits. Moreover, if the next message is sent before a previous message has been read by tk_wai_flg, the previous message will be lost. i.e., message queuing is not possible.

Since setting waiptn = 0 will result in an E_PAR error, it is guaranteed that the waiptn of tasks waiting for an event flag will not be 0. The result is that if tk_set_flg sets all bits of an event flag to 1, the task at the head of the queue will always be released from waiting no matter what its wait condition is.

The ability to have multiple tasks wait for the same event flag is useful in situations like the following. Suppose, for example, that Task B and Task C are waiting for tk_wai_flg calls (2) and (3) until Task A issues (1) tk_set_flg. If multiple tasks are allowed to wait for the event flag, the result will be the same regardless of the order in which system calls (1)(2)(3) are executed (see Figure 14). On the other hand, if multiple task waiting is not allowed and system calls are executed in the order (2), (3), (1), an E_OBJ error will result from the execution of (3) tk_wai_flg.

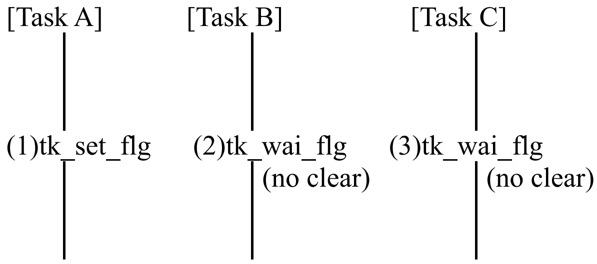


Figure 14: Multiple Tasks Waiting for One Event Flag

[Rationale for the Specification]

The reason for returning E_PAR error for specifying waiptn = 0 is that if waiptn = 0 were allowed, it would not be possible to get out of WAIT state regardless of the subsequent event flag values.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified event flag cannot be accessed due to access protection, E_DACV is returned.

Reference Event Flag Status tk_ref_flg

tk_ref_flg: Refer EventFlag Status

[C Language Interface]

ER ercd = tk_ref_flg (ID flgid, T_RFLG *pk_rflg) ;

[Parameters]

ID	flgid	EventFlagID	Event flag ID
T_RFLG* pk_rflg	Packet to Refer Eventflag		Address of packet for returning status information

[Return Parameters]

ER	ercd	ErrorCode	Error code

pk_rflg detail:

VP	exinf	ExtendedInformation	Extended information
ID	wtsk	WaitTaskInformation	Waiting task information
UINT	flgptn	EventFlagBitPattern	Event flag bit pattern

——(Other implementation-dependent parameters may be added beyond this point.)——

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (flgid is invalid or cannot be used)
E_NOEXS	Object does not exist (the event flag specified in flgid does not exist)
E_PAR	Parameter error (the address of the return parameter packet cannot be used)
E_DACV	Access Protection Violation

[Description]

References the status of the event flag specified in flgid, passing in the return parameters the current flag pattern (flgptn), waiting task information (wtsk), and extended information (exinf).

wtsk returns the ID of a task waiting for this event flag. If more than one task is waiting (only when the TA_WMUL was specified), the ID of the first task in the queue is returned. If there are no waiting tasks, wtsk = 0 is returned.

If the specified event flag does not exist, error code E_NOEXS is returned.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified event flag cannot be accessed due to access protection, E_DACV is returned.

4.4.3 Mailbox

A mailbox is an object used to achieve synchronization and communication by passing messages in system (shared) memory space. Functions are provided for creating and deleting a mailbox, sending and receiving messages in a mailbox, and referencing the mailbox status. A mailbox is an object identified by an ID number called the mailbox ID.

A mailbox has a message queue for sent messages, and a task queue for tasks waiting to receive messages. At the message sending end (making event notification), messages to be sent go in the message queue. On the message receiving end (waiting for event notification), a task fetches one message from the message queue. If there are no queued messages, the task goes to a state of waiting to receive a message from the mailbox until the next message is sent. Tasks waiting for message receipt from a mailbox are put in the task queue of that mailbox.

Since the contents of messages using this function are in memory shared by the sending and receiving sides, only the start address of a message located in this shared space is actually sent and received. The contents of the messages themselves are not copied. T-Kernel manages messages in the message queue by means of a link list. At the beginning of a message to be sent, an application program must allocate space for link list use by T-Kernel. This area is called the message header. The message header and the message body together are called a message packet. When a system call sends a message to a mailbox, the start address of the message packet (pk_msg) is passed in a parameter.

When a system call receives a message from a mailbox, the start address of the message packet is passed in a return parameter.

If messages are assigned a priority in the message queue, the message priority (msgpri) of each message must be specified in the message header (see Figure 15).

The user puts the message contents not at the beginning of the packet but after the header part (msgcont in the figure).

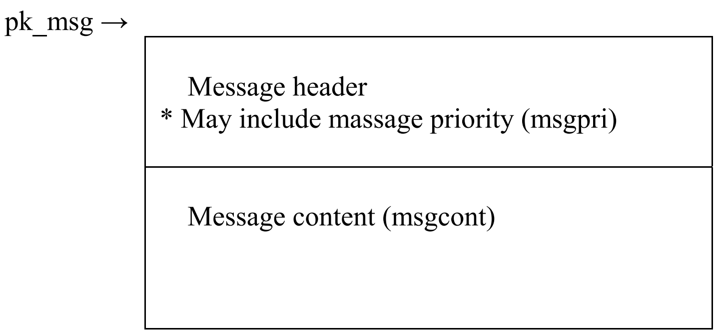


Figure 15: Format of Messages Using a Mailbox

T-Kernel overwrites the contents of the header when a message is put in the message queue (except for the message priority area). An application, on the other hand, must not overwrite the header of a message in the queue (including the message priority area). The behavior if an application overwrites the message header is not defined. This restriction applies not only to the direct writing of a message header by an application program, but also to the passing of a header address to T-Kernel and having T-Kernel overwrite the message header with the contents. Accordingly, the behavior when a message already in the message queue is again sent to a mailbox is undefined.

In SMP T-Kernel, access protection is applied to system calls that specify a Mailbox ID.

Service calls with different specifications from the T-Kernel 1.00 Specification are summarized in the table below. For more details refer to the explanation for each system call.

Call Name	Function	Different from T-Kernel 1.00 Specification
tk_cre_mbx	Create Mailbox	×
tk_del_mbx	Delete Mailbox	△
tk_snd_mbx	Send Message to Mailbox	△
tk_rcv_mbx	Receive Message from Mailbox	△

tk_ref_mbx	Reference Mailbox Status	△

Different from T-Kernel 1.00 Specification ○:No ×:Yes △: Different only in that E_DACV is returned due to the access protection

[Additional Notes]

Since the application program allocates the message header space for this mailbox function, there is no limit on the number of messages that can be queued. A system call sending a message does not enter WAIT state.

Memory blocks can be allocated dynamically from a fixed-size memory pool a variable-size memory pool or a statically allocated area can be used for message packets; however, these must not be located in task space.

Generally, a sending task allocates a memory block from a memory pool, sending that as a message packet. After a task on the receiving end receives the message, it returns the memory block directly to its memory pool.

tk_cre_mbx: Create Mailbox

[C Language Interface]

```
ID mbxid = tk_cre_mbx ( T_CMBX* pk_cmbx ) ;
```

[Parameters]

T_CMBX* pk_cmbx	Packet to Create Mailbox	Information about the mailbox to be created

pk_cmbx detail:

VP	exinf	ExtendedInformation	Extended information
ATR	mbxatr	MailboxAttribute	Mailbox attributes
ID	domid	DomainID	Domain ID
UB	oname[8]	Object name	DS Object name

——(Other implementation-dependent parameters may be added beyond this point.)——

[Return Parameters]

ID	mbxid	MailboxID	Mailbox ID
	or	ErrorCode	Error Code

[Error Codes]

E_NOMEM	Insufficient memory (memory for the control block or buffer cannot be allocated)
E_LIMIT	Number of mailboxes exceeds the system limit
E_RSATR	Reserved attribute (mbxatr is invalid or cannot be used)
E_PAR	Parameter error (pk_cmbx is invalid)
E_ID	Invalid ID number (domid is invalid or cannot be used)
E_NOEXS	Object does not exist (domain of domid does not exist)
E_ONAME	Specified object name has already been used

[Description]

Creates a mailbox, assigning it to a mailbox ID. This system call allocates a control block, etc. for the created mailbox.

exinf can be used freely by the user to store miscellaneous information about the created mailbox. The information set in this parameter can be referenced by tk_ref_mbx. If a larger area is needed for indicating user information, or if the information needs to be changed after the mailbox is created, this can be done by allocating separate memory for this purpose and putting the memory packet address in exinf. The OS pays no attention to the contents of exinf.

mbxatr indicates system attributes in its low bits and implementation-dependent information in the high bits. The system attributes part of mbxatr is as follows.

mbxatr:= (TA_TFIFO || TA_TPRI) | (TA_MFIFO || TA_MPRI) | [TA_ONAME] | [TA_NODISWAI]
| [TA_DOMID] | [(TA_PROTECTED || TA_PRIVATE || TA_PUBLIC)]

TA_TFIFO	Tasks are queued in FIFO order
TA_TPRI	Tasks are queued in priority order
TA_MFIFO	Messages are queued in FIFO order
TA_MPRI	Messages are queued in priority order
TA_ONAME	Specifies DS Object name
TA_NODISWAI	Wait disabling by tk_dis_wai is prohibited
TA_DOMID	Specifies the domain to which the task belongs
TA_PROTECTED	Sets the access protection attribute to protect
TA_PRIVATE	Sets the access protection attribute to private
TA_PUBLIC	Sets the access protection attribute to public

The queuing order of tasks waiting for a mailbox can be specified in TA_TFIFO or TA_TPRI. If the attribute is TA_TFIFO, tasks are ordered by FIFO, whereas TA_TPRI specifies queuing of tasks in order of their priority setting.

TA_MFIFO and TA_MPRI are used to specify the order of messages in the message queue (messages waiting to be received). If the attribute is TA_MFIFO, messages are ordered by FIFO; TA_MPRI specifies queuing of messages in order of the priority. Message priority is set in a special field in the message packet. Message priority is specified by positive values, with 1 indicating the highest priority and higher numbers indicating successively lower priority. The largest value that can be expressed in the PRI type is the lowest priority. Messages having the same priority are ordered as FIFO.

When TA_ONAME is specified, oname is valid and specifies the object name. When TA_ONAME is not specified, the object name is not set. The object name must be unique within the domain to which the mailbox belongs. When an object name that has already been used with another mailbox is specified, E_ONAME is returned. When the length of the character string specified for oname is 0 (initial character is termination 0), the object name is not set regardless of the specification of TA_ONAME.

When TA_DOMID is specified, domid is valid, and the domain of domid is set as the domain to which it belongs. When TA_DOMID is not specified, domid is ignored and is the domain to which the kernel domain belongs.

TA_PROTECTED, TA_PRIVATE, and TA_PUBLIC specify the access protection attribute of the mailbox. When either of the access protection attributes is not specified, the access protection is set to the public attribute. In the combination of the domain to which the task belongs and the access protection attribute, mailboxes that invoking tasks cannot access due to access protection cannot be created. When the corresponding specification is done, E_PAR is returned.

```
#define TA_TFIFO        0x00000000   /* manage task queue by FIFO */
#define TA_TPRI         0x00000001   /* manage task queue by priority */
#define TA_MFIFO        0x00000000   /* manage message queue by FIFO */
#define TA_MPRI         0x00000002   /* manage message queue by priority */
#define TA_ONAME        0x00000040   /* DS Object name */
#define TA_NODISWAI     0x00000080   /* reject wait disabling */
#define TA_DOMID        0x00010000   /* specify the domain */
#define TA_PRIVATE      0x00040000   /* set the protection attribute to private */
#define TA_PROTECTED    0x00080000   /* set the protection attribute to protect */
#define TA_PUBLIC       0x00000000   /* set the protection attribute to public*/
```

[Additional Notes]

The body of a message passed by the mailbox function is located in system (shared) memory; only its start address is actually sent and received. For this reason, a message must not be located in task space.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- TA_DOMID, TA_PROTECTED, TA_PRIVATE, and TA_PUBLIC were added to the mailbox attribute, and the domain to which it belongs and the access protection attribute are specifiable.
- The DS Object name was abolished, and replaced by the establishment of the object name. While the former was a name for debugging, the latter is a name which can be used in general for searching domain ID's, etc. The object name cannot use the same name with the same type of object in the same domain.

Delete mailbox tk_del_mbx

tk_del_mbx:Delete Mailbox

[C Language Interface]

```
ER ercd = tk_del_mbx ( ID mbxid ) ;
```

[Parameters]

ID	mbxid	MailboxID	Mailbox ID

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mbxid is invalid or cannot be used)
E_NOEXS	Object does not exist (the mailbox specified in mbxid does not exist)
E_DACV	Access Protection Violation

[Description]

Deletes the mailbox specified in mbxid.

Issuing this system call releases the mailbox ID and control block memory space, etc., associated with the mailbox.

This system call completes normally even if there are tasks waiting for messages in the deleted mailbox, but error code E_DLT is returned to each of the tasks waiting on this mailbox. Even if there are messages in the mailbox, it is deleted without returning an error code.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified mailbox cannot be accessed due to access protection, E_DACV is returned.

tk_snd_mbx: Send Message to Mailbox

[C Language Interface]

```
ER ercd = tk_snd_mbx ( ID mbxid, T_MSG *pk_msg ) ;
```

[Parameters]

ID	mbxid	MailboxID	Mailbox ID
T_MSG*	pk_msg	Packet of Message	Message packet address

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mbxid is invalid or cannot be used)
E_NOEXS	Object does not exist (the mailbox specified in mbxid does not exist)
E_PAR	Parameter error (pk_msg is a value that cannot be used)
E_DACV	Access Protection Violation

[Description]

Sends the message packet having pk_msg as its start address to the mailbox specified in mbxid. The message packet contents are not copied; only the start address (pk_msg) is passed at the time of message receipt.

If tasks are already waiting for messages in the same mailbox, the WAIT state of the task at the head of the queue is released, and the pk_msg passed to tk_snd_mbx is sent to that task, becoming a parameter returned by tk_rcv_mbx. If there are no tasks waiting for messages in the specified mailbox, the sent message goes in the message queue of that mailbox. In neither case does the task issuing tk_snd_mbx enter WAIT state.

pk_msg is the start address of the packet containing the message, including its header. The message header has the following format.

```
typedef struct t_msg {
    ?          ?          /* Impleme ntation-dep endent cont ents (fixed size) */
} T_MSG;
typedef struct t_msg_pri {
        T_MSG       msgque;     /* message queue area */
        PRI         msgpri;     /* message priority */
} T_MSG_PRI;
```

The message header is T_MSG (if TA_MFIFO attribute is specified) or T_MSG_PRI (if TA_MPRI attribute is specified). In either case, the message header has a fixed size, which can be obtained by size of (T_MSG) or size of (T_MSG_PRI).

The actual message must be put in the area after the header. There is no limit on message size, which may be of variable length.

Access protection is applied to this system call.

[Additional Notes]

Messages are sent by tk_snd_mbx regardless of the status of the receiving tasks. In other words, message sending is asynchronous. What waits in the queue is not the task itself, but the sent message. So while there are queues of waiting messages and receiving tasks, the sending task does not go to WAIT state.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified mailbox cannot be accessed due to access protection, E_DACV is returned.

Receive Message from Mailbox tk_rcv_mbx

tk_rcv_mbx: Receive Message from Mailbox

[C Language Interface]

```
ER ercd = tk_rcv_mbx ( ID mbxid, T_MSG **ppk_msg, TMO tmout ) ;
```

[Parameters]

ID	mbxid	MailboxID	Mailbox ID
TMO	tmout	Timeout	timeout

[Return Parameters]

ER	ercd	ErrorCode	Error code
T_MSG*	pk_msg	Packet of Message	Start address of message packet

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mbxid is invalid or cannot be used)
E_NOEXS	Object does not exist (the mailbox specified in mbxid does not exist)
E_PAR	Parameter error (tmout <= (-2))
E_DLT	The object being waited for was deleted (the mailbox was deleted while waiting)
E_RLWAI	Wait state released (tk_rel_wai received in wait state)
E_DISWAI	Wait released by wait disabled state
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion or in dispatch disabled state)
E_DACV	Access Protection Violation

[Description]

Receives a message from the mailbox specified in mbxid.

If no messages have been sent to the mailbox (the message queue is empty), the task issuing this system call enters WAIT state and is queued for message arrival. If there are messages in the mailbox, the task issuing this system call fetches the first message in the message queue and passes it in the return parameter pk_msg.

A maximum wait time (timeout) can be set in tmout. If the tmout time elapses before the wait release condition is met (no message arrives), the system call terminates, returning timeout error code E_TMOUT.

Only positive values can be set in tmout. The time base for tmout (time unit) is the same as that for system time (= 1 ms).

When TMO_POL = 0 is set in tmout, this means 0 was specified as the timeout value, and E_TMOUT is returned without entering WAIT state even if no message arrives. When TMO_FEVR = (-1) is set in tmout, this means infinity was specified as the timeout value, and the task continues to wait for message arrival without timing out.

Access protection is applied to this system call.

[Additional Notes]

pk_msg is the start address of the packet containing the message, including header. The message header is T_MSG (if TA_MFIFO attribute is specified) or T_MSG_PRI (if TA_MPRI attribute is specified).

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified mailbox cannot be accessed due to access protection, E_DACV is returned.

tk_ref_mbx: Refer Mailbox Status

[C Language Interface]

ER ercd = tk_ref_mbx (ID mbxid, T_RMBX *pk_rmbx) ;

[Parameters]

ID	mbxid	MailboxID	Mailbox ID
T_RMBX*	pk_rmbx	Packet to Refer Mailbox	Address of packet for returning status information

[Return Parameters]

ER	ercd	ErrorCode	Error code

pk_rmbxdetai:

VP	exinf	ExtendedInformation	Extended information
ID	wtsk	WaitTaskInformation	Waiting task information
T_MSG*	pk_msg	Packet of Message	Start address of next message packet to be received

──(Other implementation-dependent parameters may be added beyond this point.)──

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mbxid is invalid or cannot be used)
E_NOEXS	Object does not exist (the mailbox specified in mbxid does not exist)
E_PAR	Parameter error (the return parameter packet address cannot be used)
E_DACV	Access Protection Violation

[Description]

References the status of the mailbox specified in mbxid, passing in the return parameters the next message to be received (the first message in the message queue), waiting task information (wtsk), and extended information (exinf).

wtsk indicates the ID of a task waiting for the mailbox. If there are multiple waiting tasks, the ID of the first task in the queue is returned. If there are no waiting tasks, wtsk = 0 is returned.

If the specified mailbox does not exist, error code E_NOEXS is returned.

pk_msg indicates the message that will be received the next time tk_rcv_mbx is issued. If there are no messages in the message queue, pk_msg = NULL is returned. At least one of pk_msg = NULL and wtsk = 0 is always true for this system call.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified mailbox cannot be accessed due to access protection, E_DACV is returned.

4.5 Extended Synchronization and Communication Functions

Extended synchronization and communication functions use objects independent of tasks to realize more sophisticated synchronization and communication between tasks. The functions specified here include mutex, message buffer, and rendezvous port functions.

4.5.1 Mutex

A mutex is an object for mutual exclusion control among tasks when using shared resources. A mutex can change the current priority of a task along with the operation of mutex to prevent reversals in priorities without ceilings due to exclusive control.

Functions are provided for creating and deleting a mutex, locking and unlocking a mutex, and referencing mutex status. A mutex is identified by an ID number called a mutex ID.

A mutex has a status (locked or unlocked) and a queue for tasks waiting to lock the mutex. For each mutex, T-Kernel keeps track of the tasks locking it; and for each task, it keeps track of the mutexes it has locked. Before a task uses a resource, it locks a mutex corresponding to that resource. If the mutex is already locked by another task, the task waits for the mutex to become unlocked. Tasks in mutex lock waiting state are put in the mutex queue. When a task finishes with a resource, it unlocks the mutex.

For each mutex, T-Kernel keeps track of the tasks locking it; and for each task, it keeps track of the mutexes it has locked. When a task terminates and there are mutexes still locked by that task, all the mutexes are unlocked. The order in which multiple locked mutexes are unlocked is implementation-dependent. See the description of tk unl mtx for the specific processing involved.

Priority inheritance mutexes and priority ceiling mutexes are supported, as tools for managing the problem of unbounded priority inversion that can occur in mutual exclusion control. A mutex with TA_INHERIT(= 0x02) specified as mutex attribute supports priority inheritance protocol, while one with TA_CEILING(= 0x03) specified supports priority ceiling protocol.

When a priority ceiling mutex is created, a ceiling priority is assigned to it, indicating the base priority of the task having the highest base priority among the tasks able to lock that mutex. If a task having a higher base priority than the ceiling priority of the mutex tries to lock it, error code E_ILUSE is returned. If tk_chg_pri is issued in an attempt to set the base priority of a task locking a priority ceiling mutex to a value higher than the ceiling priority of that mutex, E_ILUSE is returned by the tk_chg_pri system call.

When these protocols are used, unbounded priority inversion is prevented by changing the current priority of the task in a mutex operation. Strict adherence to the priority inheritance protocol and priority ceiling protocol requires that the task current priority must always be changed to match the peak value of the following priorities. This is called strict priority control.

- The task base priority.
- When tasks lock priority inheritance mutexes, the current priority of the task having the highest current priority of the tasks waiting for those mutexes.
- When tasks lock priority ceiling mutexes, the ceiling priority of the mutex having the highest ceiling priority among those mutexes.

Note that when the current priority of a task waiting for a priority inheritance mutex changes as the result of a base priority change brought about by mutex operation or tk_chg_pri, it may be necessary to change the current priority of the task locking that mutex. This is called dynamic priority inheritance. Further, if this task is waiting for another priority inheritance mutex, dynamic priority inheritance processing may be necessary also for the task locking that mutex.

The T-Kernel specification defines, in addition to the above strict priority control, a simplified priority control limiting the situations in which the current priority is changed. The choice between the two is an implementation-dependent matter. In the simplified priority control, whereas all changes in the direction of raising the task current priority are carried out, changes in the direction of lowering that priority are made only when a task is no longer locking any mutexes. (In this case, the task current priority reverts to the base priority.) More specifically, processing to change the current priority is needed only in the following circumstances.

- When a task with a higher current priority than that of the task locking a priority inheritance mutex starts waiting for that mutex.
- When a task waiting for a priority inheritance mutex is changed to a higher current priority than that of the task locking that mutex.
- When a task locks a priority ceiling mutex having a higher ceiling priority than the task's current priority.
- When a task is no longer locking any mutexes.

When the current priority of a task is changed in connection with a mutex operation, the following processing is performed.

If the task whose priority changed is in a run state, the task precedence is changed in accordance with the new priority. Its precedence among other tasks having the same priority is implementation-dependent. Likewise, if the task whose priority changed is waiting in a queue of some kind, its order in that queue is changed based on its new priority. Its order among other tasks having the same priority is implementation-dependent.

In SMP T-Kernel, access protection is applied to system calls that specify a Mutex ID.
Service calls with different specifications from the T-Kernel 1.00 Specification are summarized in the table below. For more details refer to the explanation for each system call.

Call Name	Function	Different from T-Kernel 1.00 Specification
tk_cre_mtx	Create Mutex	×
tk_del_mtx	Delete Mutex	△
tk_loc_mtx	Lock Mutex	△
tk_unl_mtx	Unlock Mutex	△
tk_ref_mtx	Reference Mutex Status	△

Different from T-Kernel 1.00 Specification ○:No ×:Yes △:Different only in that E_DACV is returned due to the access protection

[Additional Notes]
A mutex which does not change the current task priority (TA TFIFO attribute or TA TPRI attribute) has functionality equivalent to that of a semaphore with a maximum of one resource (binary semaphore). The main differences are that a mutex can be unlocked only by the task that locked it, and a mutex is automatically unlocked when the task locking it terminates.
The term “priority ceiling protocol” is used here in a broad sense. The protocol described here is not the same as the algorithm originally proposed. Strictly speaking, it is what is otherwise referred to as a highest locker protocol or by other names.
When the change in current priority of a task due to a mutex operation results in that task’s order being changed in a priority-based queue, it may be necessary to release the waiting state of other tasks waiting for that task or for that queue.

[Rationale for the Specification]
The precedence of tasks having the same priority as the result of a change in task current priority in a mutex operation is left as implementation-dependent, for the following reason. Depending on the application, the mutex function may lead to frequent changes in current priority. It would not be desirable for this to result in constant task switching, which is what would happen if the precedence were made the lowest each time among tasks of the same priority. Ideally task precedence rather than priority should be inherited, but that results in large overhead in implementation. This aspect of the specification is therefore made an implementation-dependent matter.

Create Mutex tk_cre_mtx

tk_cre_mtx: Create Mutex

[C Language Interface]

```
ID mtxid = tk_cre_mtx ( T_CMTX *pk_cmtx ) ;
```

[Parameters]

T_CMTX* pk_cmtx Information about the mutex to be created

pk_cmtx detail:

VP	exinf	Extended information	Extended information
ATR	mtxatr	Mutex attribute	Mutex attributes
PRI	ceilpri	Upper limit priority of mutex	Mutex attributes
ID	domid	DomainID	Domain ID
UB	oname[8]	Object name	DS Object name

──(Other implementation-dependent parameters may be added beyond this point.)──

[Return Parameters]

ID	mtxid	Mutex ID or Error Code

[Error Codes]

E_NOMEM	Insufficient memory (memory for control block cannot be allocated)
E_LIMIT	Number of mutex exceeds the system limit
E_RSATR	Reserved attribute (mtxatr is invalid or cannot be used)
E_PAR	Parameter error (pk_cmtx or ceilpri is invalid)
E_ID	Invalid ID number (domid is invalid or cannot be used)
E_NOEXS	Object does not exist (domain of domid does not exist)
E_ONAME	Specified object name has already been used

[Description]

Creates a mutex, assigning it to a mutex ID.

exinf can be used freely by the user to store miscellaneous information about the created mutex. The information set in this parameter can be referenced by tk_ref_mtx. If a larger area is needed for indicating user information, or if the information needs to be changed after the mutex is created, this can be done by allocating separate memory for this purpose and putting the memory packet address in exinf. The OS pays no attention to the contents of exinf.

mtxatr indicates system attributes in its low bits and implementation-dependent information in the high bits. The system attributes part of mtxatr is as follows.

mtxatr:= (TA_TFIFO || TA_TPRI || TA_INHERIT || TA_CEILING) | [TA_ONAME] | [TA_NODISWAI]
| [TA_DOMID] | [(TA_PROTECTED || TA_PRIVATE || TA_PUBLIC)]

TA_TFIFO	Tasks are queued in FIFO order
TA_TPRI	Tasks are queued in priority order
TA_INHERIT	Priority inheritance protocol
TA_CEILING	Priority ceiling protocol
TA_ONAME	Specifies DS Object name
TA_NODISWAI	Wait disabling by tk_dis_wai is prohibited
TA_DOMID	Specifies the domain to which the task belongs
TA_PROTECTED	Sets the access protection attribute to protect
TA_PRIVATE	Sets the access protection attribute to private
TA_PUBLIC	Sets the access protection attribute to public

When the TA_TFIFO attribute is specified, the order of the mutex task queue is FIFO. If TA_TPRI, TA_INHERIT, or TA_CEILING is specified, tasks are ordered by their priority. TA_INHERIT indicates that priority inheritance protocol is used, and TA_CEILING specifies priority ceiling protocol.

Only when TA_CEILING is specified does ceilpri have validity, setting the mutex ceiling priority.

When TA_ONAME is specified, oname is valid and is set as the object name. When TA_ONAME is not specified, the object name is not set. The object name must be unique within the domain to which the mutex belongs. When an object name that has already been used with another mutex is specified, E_ONAME is returned. When the length of the character string specified for oname is 0 (initial character is termination 0), the object name is not set regardless of the specification of TA_ONAME.

When TA_DOMID is specified, domid is valid, and the domain of domid is set as the domain to which it belongs. When TA_DOMID is not specified, domid is ignored and is the domain to which the kernel domain belongs.

TA_PROTECTED, TA_PRIVATE, and TA_PUBLIC specify the access protection attribute of the mutex. When either of the access protection attributes is not specified, the access protection is set to the public attribute. In the combination of the domain to which the task belongs and the access protection attribute, mutexes that invoking tasks cannot access due to access protection cannot be created. When the corresponding specification is done, E_PAR is returned.

```
#define TA_TFIFO        0x00000000   /* manage task queue by FIFO */
#define TA_TPRI         0x00000001   /* manage task queue by priority */
#define TA_INHERIT      0x00000002   /* priority inheritance protocol */
#define TA_CEILING      0x00000003   /* priority ceiling protocol */
#define TA_ONAME        0x00000040   /* DS Object name */
#define TA_NODISWAI     0x00000080   /* reject wait disabling */
#define TA_DOMID        0x00010000   /* specifies the domain */
#define TA_PRIVATE      0x00040000   /* sets the protection attribute to private */
#define TA_PROTECTED    0x00080000   /* sets the protection attribute to protect */
#define TA_PUBLIC       0x00000000   /* sets the protection attribute to public*/
```

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- TA_DOMID, TA_PROTECTED, TA_PRIVATE, and TA_PUBLIC were added to the mutex attribute, and the domain to which it belongs and the access protection attribute are specifiable.
- The DS Object name was abolished, and replaced by the establishment of the object name. While the former was a name for debugging, the latter is a name which can be used in general for searching domain ID's, etc. The object name cannot use the same name with the same type of object in the same domain.

Delete Mutex tk_del_mtx

tk_del_mtx:Delete Mutex

[C Language Interface]

ER ercd = tk_del_mtx (ID mtxid) ;

[Parameters]

ID	mtxid	Mutex ID

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mtxid is invalid or cannot be used)
E_NOEXS	Object does not exist (the mutex specified in mtxid does not exist)
E_DACV	Access Protection Violation

[Description]

Deletes the mutex specified in mtxid.

Issuing this system call releases the mutex ID and control block memory space allocated to the mutex.

This system call completes normally even if there are tasks waiting to lock the deleted mutex, but error code E_DLT is returned to each of the tasks waiting on this mutex.

When a mutex is deleted, a task locking the mutex will have fewer locked mutexes. If the mutex was a priority inheritance mutex or priority ceiling mutex, it is possible that the priority of the task locking it will change as a result of its deletion.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified mutex cannot be accessed due to access protection, E_DACV is returned.

tk_loc_mtx: Lock Mutex

[C Language Interface]

```
ER ercd = tk_loc_mtx ( ID mtxid, TMO tmout ) ;
```

[Parameters]

ID	mtxid	Mutex ID
TMO	tmout	imeout

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mtxid is invalid or cannot be used)
E_NOEXS	Object does not exist (the mutex specified in mtxid does not exist)
E_PAR	Parameter error (tmout <= (-2))
E_DLT	The object being waited for was deleted (the mutex was deleted while waiting for a lock)
E_RLWAI	Wait state released (tk_rel_wai received in wait state)
E_DISWAI	Wait released by wait disabled state
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion or in dispatch disabled state)
E_ILUSE	Illegal use (multiple lock, or upper priority limit exceeded)
E_DACV	Access Protection Violation

[Description]

Locks the mutex specified in mtxid. If the mutex can be locked immediately, the task issuing this system call continue executing without entering WAIT state, and the mutex goes to locked status. If the mutex cannot be locked, the task issuing this system call enters WAIT state. That is, the task is put in the queue of this mutex.

A maximum wait time (timeout) can be set in tmout. If the tmout time elapses before the wait release condition is met, the system call terminates, returning timeout error code E_TMOUT.

Only positive values can be set in tmout. The time base for tmout (time unit) is the same as that for system time (= 1 ms).

When TMO_POL = 0 is set in tmout, this means 0 was specified as the timeout value and E_TMOUT is returned without entering WAIT state even if the resource cannot be locked. When TMO_FEVR = (-1) is set in tmout, this means infinity was specified as the timeout value, and the task continues to wait until the resource can be locked by it.

If the invoking task has already locked the specified mutex, error code E_ILUSE (multiple lock) is returned.

If the specified mutex is a priority ceiling mutex and the base priority of the invoking task is higher than the ceiling priority of the mutex, error code E_ILUSE (upper priority limit exceeded) is returned. **Base priority:** The task priority before it is automatically raised by the mutex. This is the priority last set by tk_chg_pri (including while the mutex is locked), or if tk_chg_pri has never been issued, the priority set when the task was created.

Access protection is applied to this system call.

[Additional Notes]

・ **Priority inheritance mutex** (TA_INHERIT attribute)

If the invoking task is waiting to lock a mutex and the current priority of the task currently locking that mutex is lower than that of the invoking task, the priority of the locking task is raised to the same level as the invoking task. If the wait ends before the waiting task can obtain a lock (due to timeout or some other reason), the priority of the task locking that mutex can be lowered (implementation-dependent option) to the highest of the following three priorities.

(a) The highest priority among the current priorities of tasks waiting to lock the mutex.
(b) The highest priority among all the other mutexes locked by the task currently locking this mutex.
(c) The base priority of the locking task.

・ **Priority ceiling mutex** (TA_CEILING attribute)
If the invoking task obtains a lock and its current priority is lower than the mutex ceiling priority, the priority of the invoking task is raised to the mutex ceiling priority.

[Items Concerning SMP T-Kernel]
Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified mutex cannot be accessed due to access protection, E_DACV is returned.

Unlock Mutex tk_unl_mtx

tk_unl_mtx: UnLock Mutex

[C Language Interface]

```
ER ercd = tk_unl_mtx ( ID mtxid ) ;
```

[Parameters]

ID	mtxid	Mutex ID

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mtxid is invalid or cannot be used)
E_NOEXS	Object does not exist (the mutex specified in mtxid does not exist)
E_ILUSE	Illegal use (not a mutex locked by the invoking task)
E_DACV	Access Protection Violation

[Description]

Unlocks the mutex specified in mtxid.

If there are tasks waiting to lock the mutex, the WAIT state of the task at the head of the queue for that mutex is released and that task locks the mutex.

If a mutex that was not locked by the invoking task is specified, error code E_ILUSE is returned.

Access protection is applied to this system call.

[Additional Notes]

If the unlocked mutex is a priority inheritance mutex or priority ceiling mutex, task priority must be lowered as follows.

If as a result of this operation the invoking task no longer has any locked mutexes, the invoking task priority is lowered to its base priority.

If the invoking task continues to have locked mutexes after this operation, the invoking task priority is lowered to whichever of the following priority levels is highest.

(a) The highest priority of all remaining locked mutexes.
(b) The base priority of the task.

However, whether or not priority is lowered in the case locked mutex remain is implementation dependent.

If a task terminates (goes to DORMANT state or NON-EXISTENT state) without explicitly unlocking mutexes, all its locked mutexes are automatically unlocked.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified mutex cannot be accessed due to access protection, E_DACV is returned.

tk_ref_mtx: Refer Mutex Status

[C Language Interface]

ER ercd = tk_ref_mtx (ID mtxid, T_RMTX *pk_rmtx) ;

[Parameters]

ID	mtxid	Mutex ID
T_RMTX*	pk_rmtx	Address of packet for returning status information

[Return Parameters]

ER	ercd	Error code
pk_rmtx detail:		
VP	exinf	Extended information
ID	htsk	ID of task locking the mutex
ID	wtsk	ID of first task waiting to lock the mutex

——(Other implementation-dependent parameters may be added beyond this point.)——

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mtxid is invalid or cannot be used)
E_NOEXS	Object does not exist (the mutex specified in mtxid does not exist)
E_PAR	Parameter error (the address of the return parameter packet cannot be used)
E_DACV	Access Protection Violation

[Description]

References the status of the mutex specified in mtxid, passing in the return parameters the task currently locking the mutex (htsk), the first task waiting to lock the mutex (wtsk), and extended information (exinf).

htsk returns the ID of the task locking the mutex. If no task is locking it, htsk = 0 is returned.

wtsk indicates the ID of a task waiting to lock the mutex. If there are multiple tasks waiting, the ID of the task at the head of the queue is returned. If no tasks are waiting, wtsk = 0 is returned.

If the specified mutex does not exist, error code E_NOEXS is returned.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified mutex cannot be accessed due to access protection, E_DACV is returned.

4.5.2 Message Buffer

A message buffer is an object for achieving synchronization and communication by the passing of variable-size messages. Functions are provided for creating and deleting a message buffer, sending and receiving messages using a message buffer, and referencing message buffer status. A message buffer is an object identified by an ID number called a message buffer ID.

A message buffer keeps a queue of messages waiting to be sent (send queue) and a queue of tasks waiting for message receipt (receive queue). It also has a message buffer space for holding sent messages. The message sender (the side making event notification) copies to the message buffer a message it wants to send. If there is insufficient space in the message buffer area, the message is queued for sending until enough space is available.

A task waiting to send a message to the message buffer is put in the send queue. On the message receipt side (waiting for event notification), one message is fetched from the message buffer. If the message buffer has no messages, the task enters WAIT state until the next message is sent. A task waiting for receipt from a message buffer is put in the receive queue of that message buffer.

Synchronous messaging can be realized by setting the message buffer space size to 0. In that case, both the sending task and receiving task wait for a system call to be invoked by each other, and the message is passed when both sides issue system calls.

In SMP T-Kernel, access protection is applied to system calls that specify a Message Buffer ID.

Service calls with different specifications from the T-Kernel 1.00 Specification are summarized in the table below. For more details refer to the explanation for each system call.

Call Name	Function	Different from T-Kernel 1.00 Specification
tk_cre_mbf	Create Message Buffer	×
tk_del_mbf	Delete Message Buffer	△
tk_snd_mbf	Send Message to Message Buffer	△
tk_rcv_mbf	Receive Message from Message Buffer	△
tk_ref_mbf	Get Message Buffer Status	△

Different from T-Kernel 1.00 Specification ○:No ×:Yes △: Different only in that E_DACV is returned due to the access protection

[Additional Notes]

Tasks waiting to send to a message buffer send messages in their queued order. Suppose Task A wanting to send a 40-byte message to a message buffer, and Task B wanting to send a 10-byte message, are queued in that order. If another task receives a message opening 20 bytes of space in the message buffer, Task B is still required to wait until Task A sends its message.

A message buffer is used to pass variable-size messages by copying them. It is the copying of messages that makes this function different from the mailbox function.

It is assumed that the message buffer will be implemented as a ring buffer.

Create Message Buffer tk_cre_mbf

tk_cre_mbf: Create MessageBuffer

[C Language Interface]

```
ID mbfid = tk_cre_mbf ( T_CMBF *pk_cmbf ) ;
```

[Parameters]

T_CMBF* pk_cmbf Packet to Create MessageBuffer   Information about the message buffer to be created

pk_cmbf detail:

VP	exinf	ExtendedInformation	Extended information
ATR	mbfatr	MessageBufferAttribute	Message buffer attributes
INT	bufsz	BufferSize	Message buffer size (in bytes)
INT	maxmsz	MaxMessageSize	Maximum message size (in bytes)
ID	domid	DomainID	Domain ID
UB	oname[8]	Object name	DS Object name

—(Other implementation-dependent parameters may be added beyond this point.)—

[Return Parameters]

ID	mbfid	MessageBufferID	Message buffer ID
	or	ErrorCode	Error Code

[Error Codes]

E_NOMEM	Insufficient memory (memory for control block or ring buffer area cannot be allocated)
E_LIMIT	Number of message buffers exceeds the system limit
E_RSATR	Reserved attribute (mbfatr is invalid or cannot be used)
E_PAR	Parameter error (pk_cmbf is invalid, or bufsz or maxmsz is negative or invalid)
E_ID	Invalid ID number (domid is invalid or cannot be used)
E_NOEXS	Object does not exist (domain of domid does not exist)
E_ONAME	Specified object name has already been used

[Description]

Creates a message buffer, assigning it to a message buffer ID. This system call allocates a control block to the created message buffer. Based on the information specified in bufsz, it allocates a ring buffer area for message queue use (for messages waiting to be received).

A message buffer is an object for managing the sending and receiving of variable-size messages. It differs from a mailbox (mbx) in that the contents of the variable-size messages are copied when the message is sent and received. It also has a function for putting the sending task in WAIT state when the buffer is full.

exinf can be used freely by the user to store miscellaneous information about the created message buffer. The information set in this parameter can be referenced by tk_ref_mbf. If a larger area is needed for indicating user information, or if the information needs to be changed after the message buffer is created, this can be done by allocating separate memory for this purpose and putting the memory packet address in exinf. The OS pays no attention to the contents of exinf.

mbfatr indicates system attributes in its low bits and implementation-dependent information in the high bits. The system attributes part of mbfatr is as follows.

```
mbfatr:= (TA_TFIFO ││ TA_TPRI) | [TA_ONAME] | [TA_NODISWAI]
      | [TA_DOMID] | [(TA_PROTECTED || TA_PRIVATE || TA_PUBLIC)]
```

TA_TFIFO	Waiting tasks waiting on call are queued in FIFO order
TA_TPRI	Waiting tasks are queued in priority order
TA_ONAME	Specifies DS Object name
TA_NODISWAI	Wait disabling by tk_dis_wai is prohibited
TA_DOMID	Specifies the domain to which the task belongs
TA_PROTECTED	Sets the access protection attribute to protect

TA_PRIVATE	Sets the access protection attribute to private
TA_PUBLIC	Sets the access protection attribute to public

The queuing order of tasks waiting for a message to be sent when the buffer is full can be specified in TA_TFIFO or TA_TPRI. If the attribute is TA_TFIFO, tasks are ordered by FIFO, whereas TA_TPRI specifies queuing of tasks in order of their priority setting. Messages themselves are queued in FIFO order only.

Tasks waiting for message receipt from a message buffer are likewise queued in FIFO order only.

When TA_ONAME is specified, oname is valid and is set as the object name. When TA_ONAME is not specified, the object name is not set. The object name must be unique within the domain to which the message buffer belongs. When an object name that has already been used with another message buffer is specified, E_ONAME is returned. When the length of the character string specified for oname is 0 (initial character is termination 0), the object name is not set regardless of the specification of TA_ONAME.

When TA_DOMID is specified, domid is valid, and the domain of domid is set as the domain to which it belongs. When TA_DOMID is not specified, domid is ignored and is the domain to which the kernel domain belongs.

TA_PROTECTED, TA_PRIVATE, and TA_PUBLIC specify the access protection attribute of the message buffer. When either of the access protection attributes is not specified, the access protection is set to the public attribute. In the combination of the domain to which the task belongs and the access protection attribute, message buffers that invoking tasks cannot access due to access protection cannot be created. When the corresponding specification is done, E_PAR is returned.

```
#define TA_TFIFO        0x00000000  /* manage task queue by FIFO */
#define TA_TPRI         0x00000001  /* manage task queue by priority */
#define TA_ONAME        0x00000040  /* DS Object name */
#define TA_NODISWAI     0x00000080  /* reject wait disabling */
#define TA_DOMID        0x00010000  /* specifies the domain */
#define TA_PRIVATE      0x00040000  /* sets the protection attribute to private */
#define TA_PROTECTED    0x00080000  /* sets the protection attribute to protect */
#define TA_PUBLIC       0x00000000  /* sets the protection attribute to public*/
```

[Additional Notes]

When there are multiple tasks waiting to send messages, the order in which the messages are sent when buffer space becomes available is always in their queued order.

If, for example, a Task A wanting to send a 30-byte message is queued with a Task B wanting to send a 10-byte message, in the order A-B, even if 20 bytes of message buffer space becomes available, Task B never sends its message before Task A.

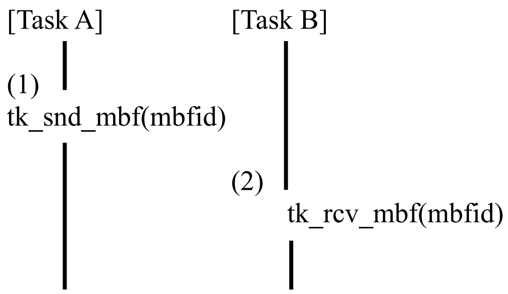
The ring buffer in which messages are queued also contains information for managing each message. For this reason, the total size of queued messages will ordinarily not be identical to the ring buffer size specified in bufsz. Normally the total message size will be smaller than bufsz. In this sense bufsz does not strictly represent the total message capacity.

It is possible to create a message buffer with bufsz = 0. In this case, communication using the message buffer is completely synchronous between the sending and receiving tasks. That is, if either tk_snd_mbf or tk_rcv_mbf is executed ahead of the other task executing the first system call goes to WAIT state. When the other system call is executed, the message is passed (copied) and both tasks resume running.

In the case of a bufsz = 0 message buffer, the specific functioning is as follows.

1)In Figure 12, Task A and Task B operate asynchronously. If Task A arrives at point (1) first and executes tk_snd_mbf (mbfid), Task A goes to send wait state until Task B arrives at point (2). If tk_ref_tsk is issued for Task A in this state, tskwait=TTW_SMBF is returned. If, on the other hand, Task B gets to point (2) first and calls tk_rcv_mbf (mbfid), Task B goes to receive wait state until Task A gets to point (1). If tk_ref_tsk is issued for Task B in this state, tskwait=TTW_RMBF is returned.

2) At the point where both Task A has executed tk_snd_mbf (mbfid) and Task B has executed tk_rcv_mbf (mbfid), a message is passed from Task A to Task B, their wait states are released and both tasks resume running.



Message send wait (TTW_SMBF) if wait entered at (1)
Message receive wait (TTW_RMBF) if wait entered at (2)

Figure 16: Synchronous Communication Using Message Buffer of bufsz =0

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- TA_DOMID, TA_PROTECTED, TA_PRIVATE, and TA_PUBLIC were added to the message buffer attribute, and the domain to which it belongs and the access protection attribute are specifiable.
- The DS Object name was abolished, and replaced by the establishment of the object name. While the former was a name for debugging, the latter is a name which can be used in general for searching domain ID's, etc. The object name cannot use the same name with the same type of object in the same domain.

Delete Message Buffer tk_del_mbf

tk_del_mbf:Delete MessageBuffer

[C Language Interface]

ER ercd = tk_del_mbf (ID mbfid) ;

[Parameters]

ID	mbfid	MessageBufferID	Message buffer ID

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mbfid is invalid or cannot be used)
E_NOEXS	Object does not exist (the message buffer specified in mbfid does not exist)
E_DACV	Access Protection Violation

[Description]

Deletes the message buffer specified in mbfid.
Issuing this system call releases the corresponding message buffer and control block memory space, as well as the message buffer space.
This system call completes normally even if there are tasks queued in the message buffer for message receipt or message sending, but error code E_DLT is returned to the tasks waiting on this message buffer. Even if there are messages left in the message buffer when it is deleted, the message buffer is deleted anyway. No error code is returned and the messages are discarded.
Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]
Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified message buffer cannot be accessed due to access protection, E_DACV is returned.

tk_snd_mbf:Send Message to MessageBuffer

[C Language Interface]

ER ercd = tk_snd_mbf (ID mbfid, VP msg, INT msgsz, TMO tmout) ;

[Parameters]

ID	mbfid	MessageBufferID	Message buffer ID
INT	msgsz	SendMessageSize	Send message size (in bytes)
VP	msg	Packet of SendMessage	Start address of send message packet
TMO	tmout	Timeout	timeout

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mbfid is invalid or cannot be used)
E_NOEXS	Object does not exist (the message buffer specified in mbfid does not exist)
E_PAR	Parameter error (msgsz <= 0, msgsz > maxmsz, value in msg can not be used, or tmout <= (-2))
E_DLT	The object being waited for was deleted (message buffer was deleted while waiting)
E_RLWAI	Wait state released (tk_rel_wai received in wait state)
E_DISWAI	Wait released by wait disabled state
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion or in dispatch disabled state)
E_DACV	Access Protection Violation

[Description]

Sends the message at the address specified in msg to the message buffer specified in mbfid. The message size is indicated in msgsz. This system call copies msgsz bytes starting from msg to the message queue of message buffer mbfid. The message queue is implemented as a ring buffer.

If msgsz is larger than the maxmsz specified with tk_cre_mbf, error code E_PAR is returned.

If there is not enough available buffer space to accommodate message msg in the message queue, the task issuing this system call goes to send wait state and is queued waiting for buffer space to become available (send queue). Waiting tasks are queued either in FIFO or priority order, depending on the setting specified at message buffer creation with tk_cre_mbf.

A maximum wait time (timeout) can be set in tmout. If the tmout time elapses before the wait release condition is met (before there is sufficient buffer space), the system call terminates, returning timeout error code E_TMOUT.

Only positive values can be set in tmout. The time base for tmout (time unit) is the same as that for system time (= 1 ms).

When TMO_POL = 0 is set in tmout, this means 0 was specified as the timeout value, and E_TMOUT is returned without entering WAIT state if there is not enough buffer space. When TMO_FEVR = (-1) is set in tmout, this means infinity was specified as the timeout value, and the task continues to wait for buffer space to become available, without timing out.

A message of size 0 cannot be sent. When msgsz <= 0 is specified, error code E_PAR is returned.

When this system call is invoked from a task-independent portion or in dispatch disabled state, error code E_CTX is returned; but in the case of tmout = TMO_POL, there may be implementations where execution from a task-independent portion or in dispatch disabled state is possible.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified message buffer cannot be accessed due to access protection, E_DACV is returned.

tk_rcv_mbf: Receive Message from MessageBuffer

[C Language Interface]

INT msgsz = tk_rcv_mbf (ID mbfid, VP msg, TMO tmout) ;

[Parameters]

ID	mbfid	MessageBufferID	Message buffer ID
VP	msg Packet	of ReceiveMessage	Start address of receive message packet
TMO	tmout	Timeout	timeout

[Return Parameters]

INT	msgsz	ReceiveMessageSize	Received message size
	or	ErrorCode	Error Code

[Error Codes]

E_ID	Invalid ID number (mbfid is invalid or cannot be used)
E_NOEXS	Object does not exist (the message buffer specified in mbfid does not exist)
E_PAR	Parameter error (value in msg cannot be used, or tmout <= (-2))
E_DLT	The object being waited for was deleted (message buffer was deleted while waiting)
E_RLWAI	Wait state released (tk_rel_wai received in wait state)
E_DISWAI	Wait released by wait disabled state
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion or in dispatch disabled state)
E_DACV	Access Protection Violation

[Description]

Receives a message from the message buffer specified in mbfid, putting it in the location specified in msg. This system call copies the contents of the first queued message in the message buffer specified in mbfid, and copies it to an area of msgsz bytes starting at address msg.

If no message has been sent to the message buffer specified in mbfid (the message queue is empty), the task issuing this system call goes to WAIT state and is put in the receive queue of the message buffer to wait for message arrival. Tasks in the receive queue are ordered by FIFO only.

A maximum wait time (timeout) can be set in tmout. If the tmout time elapses before the wait release condition is met (before a message arrives), the system call terminates, returning timeout error code E_TMOUT.

Only positive values can be set in tmout. The time base for tmout (time unit) is the same as that for system time (= 1 ms).

When TMO_POL = 0 is set in tmout, this means 0 was specified as the timeout value, and E_TMOUT is returned without entering WAIT state even if there is no message. When TMO_FEVR = (-1) is set in tmout, this means infinity was specified as the timeout value, and the task continues to wait for message arrival without timing out.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified message buffer cannot be accessed due to access protection, E_DACV is returned.

tk_ref_mbf: Get MessageBuffer Status

[C Language Interface]

```
ER ercd = tk_ref_mbf ( ID mbfid, T_RMBF *pk_rmbf ) ;
```

[Parameters]

ID	mbfid	MessageBufferID	Message buffer ID
T_RMBF*	pk_rmbf	Packet to Refer MessageBuffer	Address of packet for returning status information

[Return Parameters]

ER	ercd	ErrorCode	Error code

pk_rmbf detail:

VP	exinf	ExtendedInformation	Extended information
ID	wtsk	WaitTaskInformation	Waiting task information
ID	stsk	SendTaskInformation	Send task information
INT	msgsz	MessageSize	Size of the next message to be received (in bytes)
INT	frbufsz	FreeBufferSize	Free buffer size (in bytes)
INT	maxmsz		Maximum message size (in bytes)

——(Other implementation-dependent parameters may be added beyond this point.)——

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mbfid is invalid or cannot be used)
E_NOEXS	Object does not exist (the message buffer specified in mbfid does not exist)
E_PAR	Parameter error (the address of the return parameter packet cannot be used)
E_DACV	Access Protection Violation

[Description]

References the status of the message buffer specified in mbfid, passing in the return parameters sending task information (stsk), the size of the next message to be received (msgsz), free buffer size (frbufsz), maximum message size (maxmsz), waiting task information (wtsk), and extended information (exinf).

wtsk indicates the ID of the first task waiting to receive a message from the message buffer. The ID of the first task waiting to send to the message buffer is indicated in stsk. If multiple tasks are waiting in the message buffer queues, the ID of the task at the head of the queue is returned. If no tasks are waiting, 0 is returned.

If the specified message buffer does not exist, error code E_NOEXS is returned.

The size of the message at the head of the queue (the next message to be received) is returned in msgsz. If there are no queued messages, msgsz = 0 is returned. A message of size 0 cannot be sent.

At least one of msgsz = 0 and wtsk = 0 is always true for this system call.

frbufsz indicates the free space in the ring buffer of the message queue. This value indicates the approximate size of messages than can be sent.

maxmsz returns the maximum message size as specified with tk_cre_mbf.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified message buffer cannot be accessed due to access protection, E_DACV is returned.

4.5.3 Rendezvous Port

Rendezvous is a function for synchronization and communication between tasks, supporting the procedures for making processing requests by one task to another and for returning the processing result to the requesting task. The object for which both of these tasks wait is called a rendezvous port. The rendezvous function is typically used to realize task communication in a client/server model, but can also support more flexible synchronization and communication models.

Functions are provided for creating and deleting a rendezvous port, issuing a processing request to a rendezvous port (call rendezvous), accepting a processing request from a rendezvous port (accept rendezvous), returning the processing result (reply rendezvous), forwarding an accepted processing request to another rendezvous port (forward rendezvous to other port), and referencing rendezvous port status and rendezvous status. A rendezvous port is identified by an ID number called a rendezvous port ID.

The task issuing a processing request to a rendezvous port (the client-side task) calls a rendezvous, specifying a message (called a call message) with information about the rendezvous port, the rendezvous conditions, and the processing being requested. The task accepting a processing request on a rendezvous port (the server-side task) accepts the rendezvous, specifying the rendezvous port and rendezvous conditions.

The rendezvous conditions are indicated in a bit pattern. If the bitwise logical AND of the bit patterns on both sides (the rendezvous conditions bit pattern of the task calling a rendezvous for a rendezvous port and the rendezvous conditions bit pattern of the accepting task) is not 0, the rendezvous is established. The state of the task calling the rendezvous is WAIT on rendezvous call until the rendezvous is established. The state of the task accepting a rendezvous is WAIT on rendezvous acceptance until the rendezvous is established.

When a rendezvous is established, a call message is passed from the task that called the rendezvous to the accepting task. The state of the task calling the rendezvous goes to WAIT for rendezvous completion until the requested processing is completed. The task accepting the rendezvous is released from WAIT state and it performs the requested processing. Upon completion of the requested processing, the task accepting the rendezvous passes the result of the processing in a reply message to the calling task and ends the rendezvous. At this point the WAIT state of the task that called the rendezvous is released.

A rendezvous port has separate queues for tasks waiting on rendezvous call (call queue) and tasks waiting on rendezvous acceptance (accept queue). Note, however, that after a rendezvous is established, both tasks that formed the rendezvous are detached from the rendezvous port. In other words, a rendezvous port does not have a queue for tasks waiting for rendezvous completion nor does it keep information about the task performing the requested processing.

T-Kernel assigns an object number to identify each rendezvous when more than one is established at the same time. The rendezvous object number is called the rendezvous number. The method of assigning rendezvous numbers is implementation-dependent, but at a minimum information must be included for specifying the task that called the rendezvous. Numbers must also be uniquely assigned to the extent possible; for example, even if the same task makes multiple rendezvous calls, the first rendezvous and second rendezvous must have different rendezvous numbers assigned.

In SMP T-Kernel, access protection is applied to system calls that specify a Rendezvous Port ID.

Service calls with different specifications from the T-Kernel 1.00 Specification are summarized in the table below. For more details refer to the explanation for each system call.

Call Name	Function	Different from T-Kernel 1.00 Specification
tk_cre_por	Create Port for Rendezvous	×
tk_del_por	Delete Port for Rendezvous	△
tk_cal_por	Call Port for Rendezvous	△
tk_acp_por	Accept Port for Rendezvous	△
tk_fwd_por	Forward Rendezvous to Another Port	△
tk_rpl_rdv	Reply Rendezvous	△
tk_ref_por	Reference Rendezvous Port Status	△

Different from T-Kernel 1.00 Specification ○:No ×:Yes △: Different only in that E_DACV is returned due to the access protection

[Additional Notes]

Rendezvous is a synchronization and communication function introduced in the ADA programming language, based on Communicating Sequential Processes (CSP). In ADA, however, the rendezvous function is part of the language specification and therefore has a different role than in T-Kernel, which is a real-time kernel specification. The rendezvous ports provided by the real-time kernel are OS primitives by which an ADA rendezvous capability is implemented. Since the ADA rendezvous function differs from that in the T-Kernel specification in a number of ways, the T-Kernel-specification rendezvous functions cannot necessarily be used to implement the ADA rendezvous.

Rendezvous operation is explained here using the example in Figure 17. In this figure Task A and Task B are running asynchronously.

- If Task A first calls tk_cal_por, Task A goes to WAIT state until Task B calls tk_acp_por. The state of Task A at this time is WAIT on rendezvous call (a).
- If, on the other hand, Task B first calls tk_acp_por, Task B goes to WAIT state until Task A calls tk_cal_por. The state of Task B at this time is WAIT on rendezvous acceptance (b).
- A rendezvous is established when both Task A has called tk_cal_por and Task B has called tk_acp_por. At this time Task A remains in WAIT state while the WAIT state of Task B is released. The state of Task A is WAIT for rendezvous completion.
- The Task A WAIT state is released when Task B calls tk_rpl_rdv. Thereafter both tasks enter a run state.

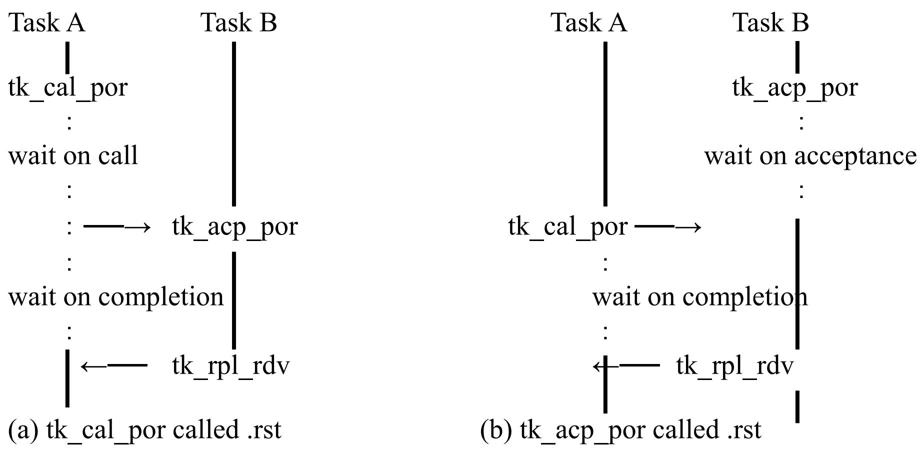


Figure 17: Rendezvous Operation

As an example of a specific method for assigning rendezvous object numbers, the ID number of the task calling the rendezvous can go in the low bits of the rendezvous number, with the high bits used for a sequential number.

[Rationale for the Specification]

While it is true that the rendezvous functionality can be achieved through a combination of other synchronization and communication functions, better efficiency and ease of programming are achieved by having a dedicated function for cases where the communication involves an acknowledgment. One advantage of the rendezvous function is that since both tasks wait until message passing is completed, no memory space needs to be allocated for storing messages.

The reason for assigning unique rendezvous numbers even when the same task does the calling is as follows. It is possible that a task, after establishing a rendezvous and going to WAIT state for its completion, will have its WAIT state released due to timeout or forcible release by another task, then again call a rendezvous and have that rendezvous established. If the same number were assigned to both the first and second rendezvous, attempting to terminate the first rendezvous would end up terminating the second rendezvous. If separate numbers are assigned to the two rendezvous and the task in WAIT state for rendezvous completion is made to remember the number of the rendezvous for which it is waiting, error will be returned when the attempt is made to terminate the first rendezvous.

tk_cre_por: Create Port for Rendezvous

[C Language Interface]

ID porid = tk_cre_por (T_CPOR *pk_cpor) ;

[Parameters]

T_CPOR* pk_cpor	Packet to Create Port	Information about the rendezvous port to be created

pk_cpor detail:

VP	exinf	ExtendedInformation	Extended information
ATR	poratr	PortAttribute	Rendezvous port attributes
INT	maxcmsz	MaxCallMessageSize	Maximum call message size (in bytes)
INT	maxrmsz	MaxReplyMessageSize	Maximum reply message size (in bytes)
ID	domid	DomainID	Domain ID
UB	oname[8]	Object name	DS Object name

——(Other implementation-dependent parameters may be added beyond this point.)——

[Return Parameters]

ID	porid	PortID	Port ID
	or	ErrorCode	Error Code

[Error Codes]

E_NOMEM	Insufficient memory (memory for control block cannot be allocated)
E_LIMIT	Number of rendezvous ports exceeds the system limit
E_RSATR	Reserved attribute (poratr is invalid or cannot be used)
E_PAR	Parameter error (pk_cpor is invalid; maxcmsz or maxrmsz is negative or invalid)
E_ID	Invalid ID number (domid is invalid or cannot be used)
E_NOEXS	Object does not exist (domain of domid does not exist)
E_ONAME	Specified object name has already been used

[Description]

Creates a rendezvous port, assigning it to a rendezvous port ID number. A rendezvous port is an object used as OS primitive for implementing a rendezvous capability. This system call allocates a control block to the created rendezvous port.

exinf can be used freely by the user to store miscellaneous information about the created rendezvous port. The information set in this parameter can be referenced by tk_ref_por. If a larger area is needed for indicating user information, or if the information may need to be changed after the rendezvous port is created, this can be done by allocating separate memory for this purpose and putting the memory packet address in exinf. The OS pays no attention to the contents of exinf.

poratr indicates system attributes in its low bits and implementation-dependent information in the high bits. The system attributes part of poratr is as follows.

poratr:= (TA_TFIFO || TA_TPRI) | [TA_ONAME] | [TA_NODISWAI]
    | [TA_DOMID] | [(TA_PROTECTED || TA_PRIVATE || TA_PUBLIC)]

TA_TFIFO	Tasks waiting on call are queued in FIFO order
TA_TPRI	Tasks waiting on call are queued in priority order
TA_ONAME	Specifies DS Object name
TA_NODISWAI	Wait disabling by tk_dis_wai is prohibited
TA_DOMID	Specifies the domain to which the task belongs
TA_PROTECTED	Sets the access protection attribute to protect
TA_PRIVATE	Sets the access protection attribute to private
TA_PUBLIC	Sets the access protection attribute to public

TA_TFIFO and TA_TPRI attributes specify the queuing order of tasks waiting on a rendezvous call. Tasks waiting on rendezvous acceptance are queued in FIFO order only.

When TA_ONAME is specified, oname is valid and is set as the object name. When TA_ONAME is not specified, the object name is not set. The object name must be unique within the domain to which the rendezvous port belongs. When an object name that has already been used with another rendezvous port is specified, E_ONAME is returned. When the length of the character string specified for oname is 0 (initial character is termination 0), the object name is not set regardless of the specification of TA_ONAME.

When TA_DOMID is specified, domid is valid, and the domain of domid is set as the domain to which it belongs. When TA_DOMID is not specified, domid is ignored and is the domain to which the kernel domain belongs.

TA_PROTECTED, TA_PRIVATE, and TA_PUBLIC specify the access protection attribute of the rendezvous port. When either of the access protection attributes is not specified, the access protection is set to the public attribute. In the combination of the domain to which the task belongs and the access protection attribute, rendezvous ports that invoking tasks cannot access due to access protection cannot be created. When the corresponding specification is done, E_PAR is returned.

```
#define TA_TFIFO        0x00000000   /* manage task queue by FIFO */
#define TA_TPRI         0x00000001   /* manage task queue by priority */
#define TA_ONAME        0x00000040   /* DS Object name */
#define TA_NODISWAI     0x00000080   /* reject wait disabling */
#define TA_DOMID        0x00010000   /* specifies the domain */
#define TA_PRIVATE      0x00040000   /* sets the protection attribute to private */
#define TA_PROTECTED    0x00080000   /* sets the protection attribute to protect */
#define TA_PUBLIC       0x00000000   /* sets the protection attribute to public*/
```

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- TA_DOMID, TA_PROTECTED, TA_PRIVATE, and TA_PUBLIC were added to the rendezvous port attribute, and the domain to which it belongs and the access protection attribute are specifiable.
- The DS Object name was abolished, and replaced by the establishment of the object name. While the former was a name for debugging, the latter is a name which can be used in general for searching domain ID's, etc. The object name cannot use the same name with the same type of object in the same domain.

tk_del_por: Delete Port for Rendezvous

[C Language Interface]

```
ER ercd = tk_del_por ( ID porid ) ;
```

[Parameters]

ID	porid	PortID	Rendezvous port ID

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (porid is invalid or cannot be used)
E_NOEXS	Object does not exist (the rendezvous port specified in porid does not exist)
E_DACV	Access Protection Violation

[Description]

Deletes the rendezvous port specified in porid.

Issuing this system call releases the ID number and control block space allocated to the rendezvous port.

This system call completes normally even if there are tasks waiting on rendezvous acceptance (tk_acp_por) or rendezvous port call (tk_cal_por) at the specified rendezvous port, but error code E_DLT is returned to the tasks in WAIT state.

Deletion of a rendezvous port by tk_del_por does not affect tasks for which rendezvous is already established. In this case, nothing is reported to the task accepting the rendezvous (not in WAIT state), and the state of the task calling the rendezvous (WAIT for rendezvous completion) remains unchanged. When the task accepting the rendezvous issues tk_rpl_rdv, that system call will execute normally even if the port on which the rendezvous was established has been deleted.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified rendezvous port cannot be accessed due to access protection, E_DACV is returned.

tk_cal_por: Call Port for Rendezvous

[C Language Interface]

```
INT rmsgsz = tk_cal_por ( ID porid, UINT calptn, VP msg, INT cmsgsz, TMO tmout ) ;
```

[Parameters]

ID	porid	PortID	Rendezvous port ID
UINT	calptn	CallBitPattern	Call bit pattern (indicating conditions of the caller)
VP	msg	Packet of Message	Message packet address
INT	cmsgsz	CallMessageSize	Call message size (in bytes)
TMO	tmout	Timeout	Timeout

[Return Parameters]

INT	rmsgsz	ReplyMessageSize	Reply message size (in bytes)
	or	ErrorCode	Error Code

[Error Codes]

E_ID	Invalid ID number (porid is invalid or cannot be used)
E_NOEXS	Object does not exist (the rendezvous port specified in porid does not exist)
E_PAR	Parameter error (cmsgsz < 0, cmsgsz > maxcmsz, calptn = 0, value that cannot be used in msg, tmout <= (-2))
E_DLT	The object being waited for was deleted (the rendezvous port was deleted while waiting)
E_RLWAI	Wait state released (tk_rel_wai received in wait state)
E_DISWAI	Wait released by wait disabled state
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion or in dispatch disabled state)
E_DACV	Access Protection Violation

[Description]

Issues a rendezvous call for a rendezvous port.

The specific operation of tk_cal_por is as follows. A rendezvous is established if there is a task waiting to accept a rendezvous at the port specified in porid and rendezvous conditions between that task and the task issuing this call overlap. In this case, the task waiting to accept the rendezvous enters READY state while the state of the task issuing tk_cal_por is WAIT for rendezvous completion. The task waiting for rendezvous completion is released from WAIT state when the other (accepting) task executes tk_rpl_rdv. The tk_cal_por system call completes at this time.

If there is no task waiting to accept a rendezvous at the port specified in porid, or if there is a task but conditions for establishing a rendezvous are not satisfied, the task issuing tk_cal_por is placed at the end of the call queue of that port and enters WAIT state on rendezvous call. The order of tasks in the call queue is either FIFO or based on priority, depending on the attributes specified when calling tk_cre_por.

The decision on rendezvous establishment is made by checking conditions in the bit patterns acpptn of the accepting task and calptn of the calling task. A rendezvous is established if the bitwise logical AND of these two bit patterns is not 0. Parameter error E_PAR is returned if calptn is 0, since no rendezvous can be established in that case.

When a rendezvous is established, the calling task can send a message (a call message) to the accepting task. The size of the call message is specified in cmsgsz. In this operation, cmsgsz bytes starting at address msg as specified by the calling task when calling tk_cal_por are copied to address msg as specified by the accepting task when calling tk_acp_por.

Similarly, when the rendezvous completes, the accepting task may send a message (reply message) to the calling task. In this operation, the contents of a reply message specified by the accepting task when calling tk_rpl_rdv are copied to address msg as specified by the calling task when calling tk_cal_por. The size of the reply message rmsgsz is set in a return parameter of tk_cal_por parameter. The original contents of the message area passed in msg by tk_cal_por end up being overwritten by the reply message received when tk_rpl_rdv executes.

Note that it is possible that message contents will be destroyed when a rendezvous is forwarded, since an area no larger than maxrmsz starting from the address msg as specified with tk_cal_por is used as a buffer. It is therefore necessary to reserve a memory space of at least maxrmsz starting from msg, regardless of the expected size of the reply message, whenever there is any possibility that a rendezvous requested by tk_cal_por might be forwarded (See the description of tk_fwd_por for details).

Error code E_PAR is returned when cmsgsz exceeds the size maxcmsz specified with tk_cre_por. This error checking is done before a task enters WAIT state on rendezvous call; and if error is detected, the task executing tk_cal_por does not enter WAIT state.

A maximum wait time (timeout) can be set in tmout. If the tmout time elapses before the wait release condition is met (rendezvous is not established), the system call terminates returning timeout error code E_TMOUT.

Only positive values can be set in tmout. The time base for tmout (time unit) is the same as that for system time (= 1 ms).

When TMO_POL = 0 is set in tmout, this means 0 was specified as the timeout value, and E_TMOUT is returned without entering WAIT state if there is no task waiting on a rendezvous at the rendezvous port, or if the rendezvous conditions are not met.

When TMO_FEVR = (-1) is set in tmout, this means infinity was specified as the timeout value, and the task continues to wait for a rendezvous to be established without timing out.

In any case TMOUT indicates the time allowed for a rendezvous to be established, and does not apply to the time from rendezvous establishment to rendezvous completion.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified rendezvous port cannot be accessed due to access protection, E_DACV is returned.

tk_acp_por:Accept Port for Rendezvous

[C Language Interface]

INT cmsgsz = tk_acp_por (ID porid, UINT acpptn, RNO *p_rdvno, VP msg, TMO tmout) ;

[Parameters]

ID	porid	PortID	Rendezvous port ID
UINT	acpptn	AcceptBitPattern	Accept bit pattern (indicating conditions for acceptance)
VP	msg	Packet of CallMessage	Message packet address
TMO	tmout	Timeout	Timeout

[Return Parameters]

ER	ercd	ErrorCode	Error code
RNO	rdvno	RendezvousNumber	Rendezvous number
INT	cmsgsz	CallMessageSize	Call message size (in bytes)

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (porid is invalid or cannot be used, or porid is a rendezvous port of another node)
E_NOEXS	Object does not exist (the rendezvous port specified in porid does not exist)
E_PAR	Parameter error (acpptn = 0, value that cannot be used in msg, or tmout <= (-2))
E_DLT	The object being waited for was deleted (the rendezvous port was deleted while waiting)
E_RLWAI	Wait state released (tk_rel_wai received in wait state)
E_DISWAI	Wait released by wait disabled state
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion or in dispatch disabled state)
E_DACV	Access Protection Violation

[Description]

Accepts a rendezvous on a rendezvous port.

The specific operation of tk_acp_por is as follows. A rendezvous is established if there is a task queued for a rendezvous call at the port specified in porid and if rendezvous conditions of that task and the task issuing this call overlap. In this case, the task queued for a rendezvous call is removed from the queue, and its state changes from WAIT on rendezvous call to WAIT for rendezvous completion. The task issuing tk_acp_por continues executing.

If there is no task waiting to call a rendezvous at the port specified in porid, or if there is a task but conditions for establishing a rendezvous are not satisfied, the task issuing tk_acp_por will enter WAIT state on rendezvous acceptance for that port. No error results if there is already another task in WAIT state on rendezvous acceptance at this time; the task issuing tk_acp_por is placed in the accept queue. It is possible to conduct multiple rendezvous operations on the same port at the same time. Accordingly, no error results even if the next rendezvous is carried out while another task is still conducting a rendezvous (before tk_rpl_rdv is called for a previously established rendezvous) at the port specified in porid.

The decision on rendezvous establishment is made by checking conditions in the bit patterns acpptn of the accepting task and calptn of the calling task. A rendezvous is established if the bitwise logical AND of these two bit patterns is not 0. If the first task does not satisfy these conditions, each subsequent task in the call queue is checked in succession. If calptn and acpptn are assigned to the same non-zero value, rendezvous is established unconditionally. Parameter error E_PAR is returned if acpptn is 0, since no rendezvous can be established in that case. All processing before a rendezvous is established is fully symmetrical on the calling and accepting ends.

When a rendezvous is established, the calling task can send a message (a call message) to the accepting task. The contents of the message specified by the calling task are copied to an area starting from msg specified by the accepting task when tk_acp_por is called. The call message size cmsgsz is passed in a return parameter of tk_acp_por.

A task accepting rendezvous can establish more than one rendezvous at a time. That is, a task that has accepted one rendezvous using tk_acp_por may execute tk_acp_por again before executing tk_rpl_rdv on the first rendezvous. The port specified for the second tk_acp_por call at this time may be the same port as the first rendezvous or a different one. It is even possible for a task already conducting a rendezvous on a given port to execute tk_acp_por again on the same port and conduct multiple rendezvous on the same port at the same time. Of course, the calling tasks will be different in each case. The return parameter rdvno passed by tk_acp_por is used to distinguish different rendezvous when more than one has been established at a given time.

It is used as a return parameter by tk_rpl_rdv when a rendezvous completes. It is also passed as a parameter to tk_fwd_por when forwarding a rendezvous. Although the exact contents of rdvno are implementation-dependent, it is expected to include information specifying the calling task on the other end of the rendezvous.

A maximum wait time (timeout) can be set in tmout. If the tmout time elapses before the wait release condition is met (rendezvous is not established), the system call terminates returning timeout error code E_TMOUT.

Only positive values can be set in tmout. The time base for tmout (time unit) is the same as that for system time (= 1 ms).

When TMO_POL = 0 is set in tmout, this means 0 was specified as the timeout value, and E_TMOUT is returned without entering WAIT state if there is no task waiting for a rendezvous call at the rendezvous port, or if the rendezvous conditions are not met.

When TMO_FEVR = (-1) is set in tmout, this means infinity was specified as the timeout value, and the task continues to wait for a rendezvous to be established without timing out.

Access protection is applied to this system call.

[Additional Notes]

The ability to queue tasks accepting rendezvous is useful when multiple servers perform the same processing concurrently. This capability also takes advantage of the task-independent nature of ports.

If a task accepting a rendezvous terminates abnormally for some reason before completing its rendezvous (before issuing tk_rpl_rdv), the task calling for the rendezvous by issuing tk_cal_por will continue waiting indefinitely for rendezvous completion without being released. To avoid such a situation, tasks accepting rendezvous should execute a tk_rpl_rdv or tk_rel_wai call when they terminate abnormally, as well as notifying the task calling for the rendezvous that the rendezvous ended in error.

rdvno contains information specifying the calling task in the rendezvous, but unique numbers should be assigned to the extent possible. Even if different rendezvous are conducted between the same tasks, a different rdvno value should be assigned to the first and second rendezvous to avoid problems like the following.

If a task that called tk_cal_por and is waiting for rendezvous completion has its WAIT state released by tk_rel_wai or by tk_ter_tsk + tk_sta_tsk or the like, conceivably it may execute tk_cal_por a second time, resulting in establishment of a rendezvous. If the same rdvno value is assigned to the first rendezvous and the subsequent one, then if tk_rpl_rdv is executed for the first rendezvous it will end up terminating the second one. By assigning rdvno numbers uniquely and having the task in WAIT state for rendezvous completion remember the number of the expected rdvno, it will be possible to detect the error when tk_rpl_rdv is called for the first rendezvous.

One possible method of assigning rdvno numbers is to put the ID number of the task calling the rendezvous in the low bits of rdvno, using the high bits for a sequential number.

The capability of setting rendezvous conditions in calptn and acpptn can be applied to implement a rendezvous selective acceptance function like the ADA select function. A specific processing approach equivalent to an ADA select statement (Figure 14(a)) is shown in Figure 14(b).

The ADA select function is provided only on the accepting end, but it is also possible to implement a select function on the calling end by specifying multiple bits in calptn.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified rendezvous port cannot be accessed due to access protection, E_DACV is returned.

[Rationale for the Specification]

The reason for specifying separate system calls tk_cal_por and tk_acp_por even though the conditions for establishing a rendezvous mirror for each other on the calling and accepting sides is because the processing required after a rendezvous is established differs for the tasks on each side. That is, whereas the calling task enters WAIT state after the rendezvous is established, the accepting task enters READY state.

```
select
      when condition_A
            accept entry_A do ... end;
or
      when condition_B
            accept entry_B do ... end;
or
      when condition_C
            accept entry_C do ... end;
end select;
```

Figure 18(a): Using Rendezvous to Implement ADA select Function

- Rather than entry_A, entry_B, and entry_C each corresponding to one rendezvous port, the entire select statement corresponds to one rendezvous port.

- entry_A, entry_B, and entry_C correspond to calptn and acpptn bits 2^0, 2^1, and 2^2.

- A select statement in a typical ADA program looks like the following.

```
ptn := 0;
if conditon_A then ptn := ptn + 2^0 endif;
if conditon_B then ptn := ptn + 2^1 endif;
if conditon_C then ptn := ptn + 2^2 endif;
tk_acp_por(acpptn := ptn);
```

- If the program contains in addition to the select statement a simple entry_A accept with no select,

```
tk_acp_por(acpptn := 2^0);
```

can be executed. If it is required to have entry_A, entry_B, and entry_C wait unconditionally by OR logic,

```
tk_acp_por(acpptn := 2^2+2^1+2^0);
```

can be executed.

- If the call on the rendezvous calling side is for entry_A,

```
tk_cal_por(calptn := 2^0);
```

can be executed; and if the call is for entry_C,

```
tk_cal_por(calptn := 2^2);
```

can be executed.

Figure 18(b): Using Rendezvous to Implement ADA select Function

tk_fwd_por: Forward Rendezvous to Other Port

[C Language Interface]

ER ercd = tk_fwd_por (ID porid, UINT calptn, RNO rdvno, VP msg, INT cmsgsz) ;

[Parameters]

ID	porid	PortID	Rendezvous port ID
UINT	calptn	CallBitPattern	Call bit pattern (indicating conditions of the caller)
RNO	rdvno	RendezvousNumber	Rendezvous number before forwarding
VP	msg	Packet of CallMessage	Message packet address
INT	cmsgsz	CallMessageSize	Call message size (in bytes)

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (porid is invalid or cannot be used, or porid is a rendezvous port of another node)
E_NOEXS	Object does not exist (the rendezvous port specified in porid does not exist)
E_PAR	Parameter error (cmsgsz < 0, cmsgsz > maxcmsz after forwarding, cmsgsz > maxrmsz before forwarding, calptn = 0, or msg has a value that cannot be used)
E_OBJ	Invalid object state (rdvno is invalid, or maxrmsz (after forwarding) > maxrmsz (before forwarding))
E_CTX	Context error (issued from task-independent portion (implementation-dependent error))
E_DISWAI	Wait released by wait disabled state
E_DACV	Access Protection Violation

[Description]

Forward an accepted rendezvous to another rendezvous port.

The task issuing this system call (here “Task X”) must have accepted the rendezvous specified in porid; i.e., this system call can be issued only after executing tk_acp_por. In the discussion that follows, the rendezvous calling task is “Task Y”, and the rendezvous number passed in a return parameter by tk_acp_por is rdvno. After tk_fwd_por is issued in this situation, the rendezvous between Task X and Task Y is released, and all processing thereafter is the same as if Task Y had called for a rendezvous on another port (rendezvous port B) passed to this system call in porid.

The specific operations of tk_fwd_por are as follows.

1. The rendezvous specified in rdvno is released.
2. Task Y goes to WAIT state on rendezvous call for the rendezvous port specified in porid. The bit conditions representing the call select conditions in this case are not those given in the calptn specified by Task Y when it called tk_cal_por, but those in the calptn specified by Task X when it called tk_fwd_por. The state of Task Y goes from WAIT for rendezvous completion back to WAIT on rendezvous call.
3. Then if a rendezvous for the rendezvous port specified in porid is accepted, a rendezvous is established between the accepting task and Task Y. Naturally, if there is a task already waiting to accept a rendezvous on the rendezvous port specified in porid and the rendezvous conditions are met, executing tk_fwd_por will immediately cause a rendezvous to be established. Here too, as with calptn, the message sent to the accepting task when the rendezvous is established is that specified in tk_fwd_por by Task X, not that specified in tk_cal_por by Task Y.
4. After the new rendezvous has completed, the reply message returned to the calling task by tk_rpl_rdv is copied to the area specified in the msg parameter passed to tk_cal_por by Task Y, not to the area specified in the msg parameter passed to tk_fwd_por by Task X.

Essentially the following situation:

Executing tk_fwd_por (porid=portB, calptn=ptnB, msg=mesB)

after tk_cal_por (porid=portA, calptn=ptnA, msg=mesA)

is the same as the following:

Executing tk_cal_por (porid=portB, calptn=ptnB, msg=mesB).

Note that it is not necessary to log the history of rendezvous forwarding.

If tk_ref_tsk is executed for a task that has returned to WAIT on rendezvous call due to tk_fwd_por execution, the value returned in tskwait is TTW_CAL. Here wid is the ID of the rendezvous port to which the rendezvous was forwarded.

tk_fwd_por execution completes immediately; in no case does this system call go to a WAIT state. A task issuing tk_fwd_por loses any relationship to the rendezvous port on which the forwarded rendezvous was established, the forwarding destination (the port specified in porid), and the tasks conducting rendezvous on these ports.

Error code E_PAR is returned if cmsgsz is larger than maxcmsz of the rendezvous port after forwarding. This error is checked before the rendezvous is forwarded. If this error occurs, the rendezvous is not forwarded and the rendezvous specified in rdvno is not released.

The send message specified with tk_fwd_por is copied to another memory space (such as the message area specified with tk_cal_por) when tk_fwd_por is executed. Accordingly, even if the contents of the message area specified in the msg parameter passed to tk_fwd_por are changed before the forwarded rendezvous is established, the forwarded rendezvous will not be affected.

When a rendezvous is forwarded by tk_fwd_por, maxrmsz of the rendezvous port after forwarding (specified in porid) must be no larger than maxrmsz of the rendezvous port on which the rendezvous was established before forwarding. If maxrmsz of the rendezvous port after forwarding is larger than maxrmsz of the rendezvous port before forwarding, this means the destination rendezvous port was not suitable, and error code E_OBJ is returned. The task calling the rendezvous readies a reply message receiving area based on the maxrmsz of the rendezvous port before forwarding. If the maximum size for the reply message increases when the rendezvous is forwarded, this may indicate that an unexpectedly large reply message is being returned to the calling rendezvous port, which would cause problems. For this reason, a rendezvous cannot be forwarded to a rendezvous port having a larger maxrmsz.

Similarly, cmsgsz indicating the size of the message sent by tk_fwd_por must be no larger than maxrmsz of the rendezvous port on which the rendezvous was established before forwarding. This is because it is assumed that the message area specified with tk_cal_por will be used as a buffer in implementing tk_fwd_por. If cmsgsz is larger than maxrmsz of the rendezvous port before forwarding, error code E_PAR is returned (See Additional Note for details).

It is not necessary to issue tk_fwd_por and tk_rpl_rdv from a task-independent portion, but it is possible to issue these system calls from dispatch disabled or interrupts disabled state. This capability can be used to perform processing that is inseparable from tk_fwd_por or tk_rpl_rdv. Whether or not error checking is done for issuing of these system calls from a task-independent portion is implementation-dependent.

When, as a result of tk_fwd_por Task Y that was in WAIT state for rendezvous completion reverts to WAIT on rendezvous call, the timeout until rendezvous establishment is always treated as Wait forever (TMO_FEVR).

The rendezvous port being forwarded to may be the same port used for the previous rendezvous (the rendezvous port on which the rendezvous specified in rdvno was established). In this case, tk_fwd_por cancels the previously accepted rendezvous. Even in this case, however, the call message and calptn parameters are changed to those passed to tk_fwd_por by the accepting task, not those passed to tk_cal_por by the calling task.

It is possible to forward a rendezvous that has already been forwarded.

Access protection is applied to this system call. The destination rendezvous port must be accessible from both the task issuing this call and the task forwarding the call.

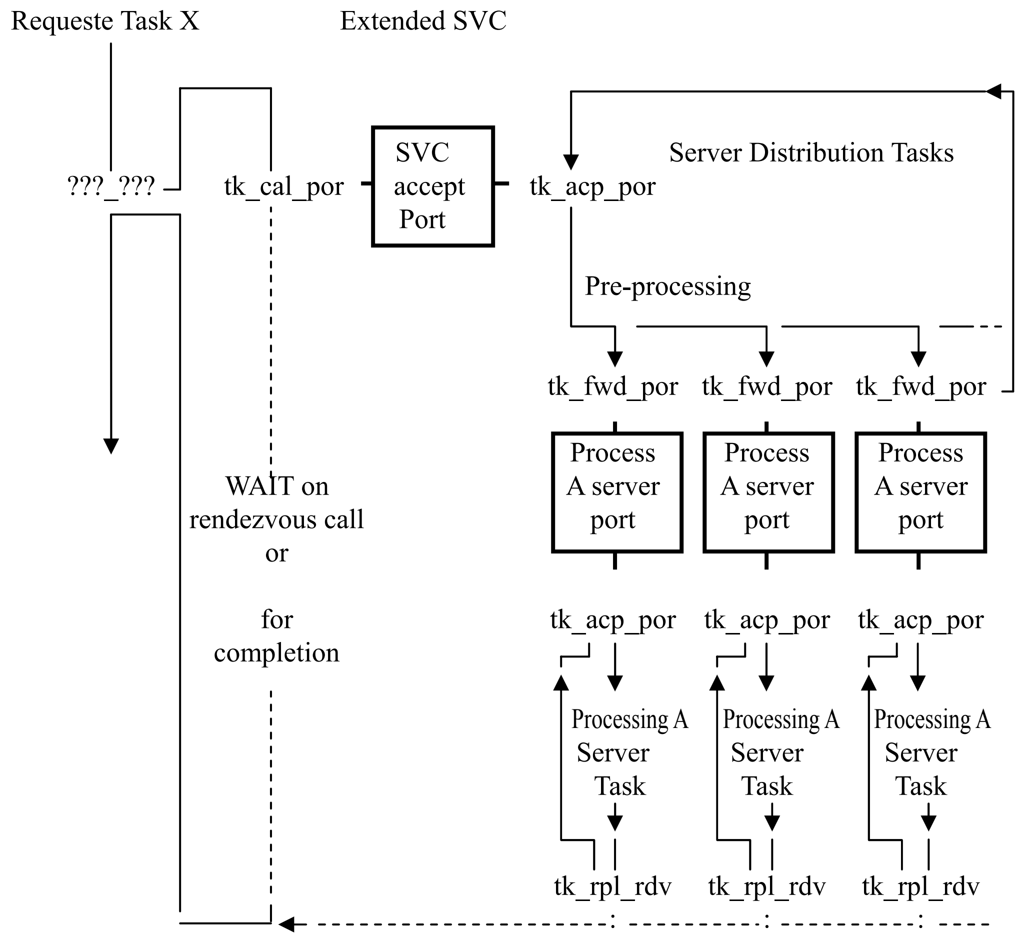
[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified rendezvous port cannot be accessed due to access protection, E_DACV is returned.

[Additional Notes]

A server task operation using tk_fwd_por is illustrated in Figure 19.



* Bold outlines indicate rendezvous ports (rendezvous entries)
* While it is possible to use tk_cal_por in place of tk_fwd_por, this results in rendezvous nesting. Assuming it is acceptable for requesting Task X to resume execution after the processing of server tasks A to C is completed, use of tk_fwd_por does away with the need for rendezvous nesting and results in more efficient operations.

Figure 19: Server Task Operation Using tk_fwd_por

Generally tk_fwd_por is executed by server distribution tasks (tasks for distributing server-accepted processing to other tasks) as shown in Figure 15. Accordingly, a server distribution task that has executed tk_fwd_por must go on to processing for acceptance of the next request regardless of whether the forwarded rendezvous is established or not. The tk_fwd_por message area in this case is used for processing the next request, making it necessary to ensure that changes to the contents of this message area will not affect the previously forwarded rendezvous. For this reason, after tk_fwd_por is executed, it must be possible to modify the contents of the message area indicated in msg passed to tk_fwd_por even before the forwarded rendezvous is established.

In order to fulfill this requirement, in implementation it is allowed to use the message area specified with tk_cal_por as a buffer. That is, in the tk_fwd_por processing, it is permissible to copy the call messages specified with tk_fwd_por to the message area indicated in msg when tk_cal_por was called, and for the task calling tk_fwd_por to change the contents of the message area. When a rendezvous is established, the message placed in the tk_cal_por message area is passed to the accepting task, regardless of whether the rendezvous is one that was forwarded from another port.

The following specifications are made to allow this sort of implementation to be used.

- If there is a possibility that a rendezvous requested by tk_cal_por may be forwarded, a memory space of at least maxrmsz bytes must be allocated starting from msg (passed to tk_cal_por), regardless of the size of the expected reply message.

- The send message size cmsgsz passed to tk_fwd_por must be no larger than maxrmsz of the rendezvous port before forwarding.

- If a rendezvous is forwarded using tk_fwd_por, maxrmsz of the destination port rendezvous must not be larger than maxrmsz of the port before forwarding.

[Rationale for the Specification]

The tk_fwd_por specification is designed not to require logging a history of rendezvous forwarding, so as to reduce the number of states that must be kept track of in the system as a whole. Applications that require such a log to be kept can use nested pairs of tk_cal_por and tk_acp_por rather than using tk_fwd_por.

tk_rpl_rdv: Reply Rendezvous

[C Language Interface]

ER ercd = tk_rpl_rdv (RNO rdvno, VP msg, INT rmsgsz) ;

[Parameters]

RNO	rdvno	RendezvousNumber	Rendezvous number
VP	msg	Packet of ReplyMessage	Reply message packet address
INT	rmsgsz	ReplyMessageSize	Reply message size (in bytes)

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_PAR	Parameter error (rmsgsz < 0, rmsgsz > maxrmsz, or value in msg cannot be used)
E_OBJ	Invalid object state (rdvno is invalid)
E_CTX	Context error (issued from task-independent portion (implementation-dependent error))

[Description]

Returns a reply to the calling task in the rendezvous, ending the rendezvous.

The task issuing this system call (here “Task X”) must be engaged in a rendezvous, i.e., this system call can be issued only after executing tk_acp_por. In the discussion that follows, the rendezvous calling task is “Task Y”, and the rendezvous number passed in a return parameter by tk_acp_por is rdvno. When tk_rpl_rdv is executed in this situation, the rendezvous state between Task X and Task Y is released, and the state of Task Y goes from WAIT for rendezvous completion back to READY state.

When a rendezvous is ended by tk_rpl_rdv, accepting Task X can send a reply message to calling Task Y. The contents of the message specified by the accepting task are copied to the memory space specified in msg passed by Task Y to tk_cal_por. The size of the reply message rmsgsz is passed as a tk_cal_por return parameter.

Error code E_PAR is returned if rmsgsz is larger than maxrmsz specified with tk_cre_por. When this error is detected, the rendezvous is not ended and the task that called tk_cal_por remains in WAIT state for rendezvous completion.

It is not necessary to issue tk_fwd_por and tk_rpl_rdv from a task-independent portion, but it is possible to issue these system calls from dispatch disabled or interrupts disabled state. This capability can be used to perform processing that is inseparable from tk_fwd_por or tk_rpl_rdv. Whether or not error checking is done for issuing of these system calls from a task-independent portion is implementation-dependent.

[Additional Notes]

If a task calling a rendezvous aborts for some reason before completion of the rendezvous (before tk_rpl_rdv is executed), the accepting task has no direct way of knowing of the abort. In such a case, error code E_OBJ is returned to the rendezvous accepting task when it executes tk_rpl_rdv.

After a rendezvous is established, tasks are in principle detached from the rendezvous port and have no need to reference information about each other. However, since the value of maxrmsz, used when checking the length of the reply message sent using tk_rpl_rdv, is dependent on the rendezvous port, the task in rendezvous must record this information somewhere. One possible implementation would be to put this information in the TCB of the calling task after it goes to WAIT state, or in another area that can be referenced from the TCB, such as a stack area.

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

[Rationale for the Specification]

The parameter rdvno is passed to tk_rpl_rdv and tk_fwd_por as information for distinguishing one established rendezvous from another, but the rendezvous port ID (porid) used when establishing a rendezvous is not specified. This is based on the design principle that tasks are no longer related to rendezvous ports after a rendezvous has been established.

Error code E_OBJ rather than E_PAR is returned for an invalid rdvno. This is because rdvno itself is an object indicating the task that called the rendezvous.

Reference Port Status tk_ref_por

tk_ref_por:Refer Port Status

[C Language Interface]

```
ER ercd = tk_ref_por ( ID porid, T_RPOR *pk_rpor ) ;
```

[Parameters]

ID	porid	PortID	Rendezvous port ID
T_RPOR*	pk_rpor	Packet to Refer Port	Start address of packet for returning status information

[Return Parameters]

ER	ercd	ErrorCode	Error code

pk_rpor detail:

VP	exinf	ExtendedInformation	Extended information
ID	wtsk	WaitTaskInformation	Waiting task information
ID	atsk	AcceptTaskInformation	Accept task information
INT	maxcmsz		Maximum call message size (in bytes)
INT	maxrmsz		Maximum reply message size (in bytes)

——(Other implementation-dependent parameters may be added beyond this point.)——

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (porid is invalid or cannot be used)
E_NOEXS	Object does not exist (the rendezvous port specified in porid does not exist)
E_PAR	Parameter error (the return parameter packet address cannot be used)
E_DACV	Access Protection Violation

[Description]

References the status of the rendezvous port specified in porid, passing in return parameters information about the accepting task (atsk), information about the task waiting on a rendezvous call (wtsk), maximum message sizes (maxcmsz, maxrmsz), and extended information (exinf).

wtsk indicates the ID of a task in WAIT state on rendezvous call at the rendezvous port. If there is no task waiting on rendezvous call, wtsk = 0 is returned. atsk indicates the ID of a task in WAIT state on rendezvous acceptance at the rendezvous port. If there is no task waiting for rendezvous acceptance, atsk = 0 is returned.

If there are multiple tasks waiting on rendezvous call or acceptance at this rendezvous port, the ID of the task at the head of the call queue and accept queue is returned.

If the specified rendezvous port does not exist, error code E_NOEXS is returned.

Access protection is applied to this system call.

[Additional Notes]

This system call cannot be used to get information about tasks involved in a currently established rendezvous.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified rendezvous port cannot be accessed due to access protection, E_DACV is returned.

4.6 Memory Pool Management Functions

Memory pool management functions provide software-based management of memory pools and memory block allocation.

There are fixed-size memory pools and variable-size memory pools, which are considered separate objects and require separate sets of system calls for their operation. Memory blocks allocated from a fixed-size memory pool are all of one fixed size, whereas memory blocks from a variable-size memory pool can be of various sizes.

The memory managed by the memory pool management functions is all in system space; there is no T-Kernel function for managing task space memory.

4.6.1 Fixed-size Memory Pool

A fixed-size memory pool is an object used for dynamic management of fixed-size memory blocks. Functions are provided for creating and deleting a fixed-size memory pool, getting and returning memory blocks in a fixed-size memory pool, and referencing the status of a fixed-size memory pool. A fixed-size memory pool is an object identified by an ID number called a fixed-size memory pool ID.

A fixed-size memory pool has a memory space used as the fixed-size memory pool (called a fixed-size memory pool area or simply memory pool area), and a queue for tasks waiting for memory block allocation. A task wanting to allocate a memory block from a fixed-size memory pool that lacks sufficient available memory space goes to WAIT state for fixed-size memory block until memory blocks are returned to the pool. A task in this state is put in the task queue of the fixed-size memory pool.

In SMP T-Kernel, access protection is applied to system calls that specify a Fixed-sized Memory Pool ID.

Service calls with different specifications from the T-Kernel 1.00 Specification are summarized in the table below. For more details refer to the explanation for each system call.

Call Name	Function	Different from T-Kernel 1.00 Specification
tk_cre_mpf	Create Fixed-size Memory Pool	×
tk_del_mpf	Delete Fixed-size Memory Pool	△
tk_get_mpf	Get Fixed-size Memory Block	△
tk_rel_mpf	Release Fixed-size Memory Block	△
tk_ref_mpf	Reference Fixed-size Memory Pool Status	△

Different from T-Kernel 1.00 Specification ○:No ×:Yes △: Different only in that E_DACV is returned due to the access protection

[Additional Notes]

When memory blocks of various sizes are needed from fixed-size memory pools, it is necessary to provide multiple memory pools of different sizes.

tk_cre_mpf: Create Fixed-size MemoryPool

[C Language Interface]

ID mpfid = tk_cre_mpf (T_CMPF *pk_cmpf) ;

[Parameters]

T_CMPF* pk_cmpf Packet to Create MemoryPool Information about the memory pool to be created

pk_cmpf detail:

VP	exinf	ExtendedInformation	Extended information
ATR	mpfatr	MemoryPoolAttribute	Memory pool attributes
INT	mpfcnt	MemoryPoolBlockCount	Memory pool block count
INT	blfsz	MemoryBlockSize	Memory block size (in bytes)
ID	domid	DomainID	Domain ID
UB	oname[8]	Object name	DS Object name

——(Other implementation-dependent parameters may be added beyond this point.)——

[Return Parameters]

ID	mpfid	MemoryPoolID	Fixed-size memory pool ID
	or	ErrorCode	Error Code

[Error Codes]

E_NOMEM	Insufficient memory (memory for control block or memory pool area cannot be allocated)
E_LIMIT	Number of fixed-size memory pools exceeds the system limit
E_RSATR	Reserved attribute (mpfatr is invalid or cannot be used)
E_PAR	Parameter error (pk_cmpf is invalid; mpfsz or blfsz is negative or invalid)
E_ID	Invalid ID number (domid is invalid or cannot be used)
E_NOEXS	Object does not exist (domain of domid does not exist)
E_ONAME	Specified object name has already been used

[Description]

Creates a fixed-size memory pool, assigning it to a fixed-size memory pool ID. This system call allocates a memory space for use as a memory pool based on the information specified in parameters mpfcnt and blfsz, and assigns a control block to the memory pool. A memory block of size blfsz can be allocated from the created memory pool by calling the tk_get_mpf system call.

exinf can be used freely by the user to store miscellaneous information about the created memory pool. The information set in this parameter can be referenced by tk_ref_mpf. If a larger area is needed for indicating user information, or if the information needs to be changed after the memory pool is created, this can be done by allocating separate memory for this purpose and putting the memory packet address in exinf. The OS pays no attention to the contents of exinf.

mpfatr indicates system attributes in its low bits and implementation-dependent information in the high bits. The system attributes part of mpfatr is as follows.

```
mpfatr:= (TA_TFIFO || TA_TPRI) | [TA_ONAME] | [TA_NODISWAI]
         | (TA_RNG0 || TA_RNG1 ||  TA_RNG2  || TA_RNG3)
         | [TA_DOMID] | [(TA_PROTECTED || TA_PRIVATE || TA_PUBLIC)]
```

TA_TFIFO	Tasks waiting for memory allocation are queued in FIFO order
TA_TPRI	Tasks waiting for memory allocation are queued in priority order
TA_RNGn	Memory access privilege is set to protection level n
TA_ONAME	Specifies DS Object name
TA_NODISWAI	Wait disabling by tk_dis_wai is prohibited
TA_DOMID	Specifies the domain to which the task belongs

TA_PROTECTED	Sets the access protection attribute to protect
TA_PRIVATE	Sets the access protection attribute is set to private
TA_PUBLIC	Sets the access protection attribute is set to public

The queuing order of tasks waiting for memory block allocation from a memory pool can be specified in TA_TFIFO or TA_TPRI. If the attribute is TA_TFIFO, tasks are ordered by FIFO, whereas TA_TPRI specifies queuing of tasks in order of their priority setting.

TA_RNGn is specified to limit the protection levels at which memory can be accessed. Only tasks running at the same or higher protection level than the one specified can access the allocated memory. If a task running at a lower protection level attempts access, a CPU protection fault exception is raised. For example, memory allocated from a memory pool specified as TA_RNG1 can be accessed by tasks running at levels TA_RNG0 or TA_RNG1, but not by tasks running at levels TA_RNG2 or TA_RNG3.

The created memory pool is in resident memory in system space. There is no T-Kernel function for creating a memory pool in task space.

When TA_ONAME is specified, oname is valid and is set as the object name. When TA_ONAME is not specified, the object name is not set. The object name must be unique within the domain to which the memory pool belongs. When an object name that has already been used with another memory pool is specified, E_ONAME is returned. When the length of the character string specified for oname is 0 (initial character is termination 0), the object name is not set regardless of the specification of TA_ONAME.

When TA_DOMID is specified, domid is valid, and the domain of domid is set as the domain to which it belongs. When TA_DOMID is not specified, domid is ignored and is the domain to which the kernel domain belongs.

TA_PROTECTED, TA_PRIVATE, and TA_PUBLIC specify the access protection attribute of the memory pool. When either of the access protection attributes is not specified, the access protection is set to the public attribute. In the combination of the domain to which the task belongs and the access protection attribute, memory pools that invoking tasks cannot access due to access protection cannot be created. When the corresponding specification is done, E_PAR is returned.

```
#define TA_TFIFO        0x00000000   /* manage task queue by FIFO */
#define TA_TPRI         0x00000001   /* manage task queue by priority */
#define TA_ONAME        0x00000040   /* DS Object name */
#define TA_NODISWAI     0x00000080   /* reject wait disabling */
#define TA_RNG0         0x00000000   /* protection level 0 */
#define TA_RNG1         0x00000100   /* protection level 1 */
#define TA_RNG2         0x00000200   /* protection level 2 */
#define TA_RNG3         0x00000300   /* protection level 3 */
#define TA_DOMID        0x00010000   /* specifies the domain */
#define TA_PRIVATE      0x00040000   /* sets the protection attribute to private */
#define TA_PROTECTED    0x00080000   /* sets the protection attribute to protect */
#define TA_PUBLIC       0x00000000   /* sets the protection attribute to public*/
```

[Additional Notes]

In the case of a fixed-size memory pool, separate memory pools must be provided for different block sizes. That is, if various memory block sizes are required, memory pools must be created for each block size.

For the sake of portability, the TA_RNGn attribute must be accepted even by a system without an MMU. It is possible, for example, to treat all TA_RNGn as equivalent to TA_RNG0; but no error must be returned.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- TA_DOMID, TA_PROTECTED, TA_PRIVATE, and TA_PUBLIC were added to the memory pool attribute, and the domain to which it belongs and the access protection attribute are specifiable.
- The DS Object name was abolished, and replaced by the establishment of the object name. While the former was a name for debugging, the latter is a name which can be used in general for searching domain ID's, etc. The object name cannot use the same name with the same type of object in the same domain.

Delete Fixed-size Memory Pool tk_del_mpf

tk_del_mpf: Delete Fixed-size MemoryPool

[C Language Interface]

```
ER ercd = tk_del_mpf ( ID mpfid ) ;
```

[Parameters]

ID	mpfid	MemoryPoolID	Fixed-size memory pool ID

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mpfid is invalid or cannot be used)
E_NOEXS	Object does not exist (the fixed-size memory pool specified in mpfid does not exist)
E_DACV	Access Protection Violation

[Description]

Deletes the fixed-size memory pool specified in mpfid.

No check or notification is made as to whether there are tasks using memory allocated from this memory pool. The system call completes normally even if some blocks have not been returned to the pool.

Issuing this system call releases the memory pool ID number, the control block memory space and the memory pool space itself.

This system call completes normally even if there are tasks waiting for memory block allocation from the deleted memory pool, but error code E_DLT is returned to the tasks in WAIT state.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified memory pool cannot be accessed due to access protection, E_DACV is returned.

Get Fixed-size Memory Pool tk_get_mpf

tk_get_mpf: Get Fixed-size Memory Block

[C Language Interface]

ER ercd = tk_get_mpf (ID mpfid, VP *p_blf, TMO tmout) ;

[Parameters]

ID	mpfid	MemoryPoolID	Fixed-size memory pool ID
TMO	tmout	Timeout	imeout

[Return Parameters]

ER	ercd	ErrorCode	Error code
VP	blf	BlockStartAddress	Memory block start address

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mpfid is invalid or cannot be used)
E_NOEXS	Object does not exist (the fixed-size memory pool specified in mpfid does not exist)
E_PAR	Parameter error (tmout <= (-2))
E_DLT	The object being waited for was deleted (the memory pool was deleted while waiting)
E_RLWAI	Wait state released (tk_rel_wai received in wait state)
E_DISWAI	Wait released by wait disabled state
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion or in dispatch disabled state)
E_DACV	Access Protection Violation

[Description]

Gets a memory block from the fixed-size memory pool specified in mpfid. The start address of the allocated memory block is returned in blf. The size of the allocated memory block is the value specified in the blfsz parameter when the fixed-size memory pool was created.

The allocated memory is not cleared to zero, and the memory block contents are indeterminate.

If a block cannot be allocated from the specified memory pool, the task that issued tk_get_blf is put in the queue of tasks waiting for memory allocation from that memory pool, and waits until memory can be allocated.

A maximum wait time (timeout) can be set in tmout. If the tmout time elapses before the wait release condition is met (memory space does not become available), the system call terminates, returning timeout error code E_TMOUT.

Only positive values can be set in tmout. The time base for tmout (time unit) is the same as that for system time (= 1 ms).

When TMO_POL = 0 is set in tmout, this means 0 was specified as the timeout value, and E_TMOUT is returned without entering WAIT state even if memory cannot be allocated.

When TMO_FEVR =(-1) is set in tmout, this means infinity was specified as the timeout value, and the task continues to wait for memory allocation without timing out.

The queuing order of tasks waiting for memory block allocation is either FIFO or in the order of task priority, depending on the memory pool attribute.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified memory pool cannot be accessed due to access protection, E_DACV is returned.

tk_rel_mpf: Release Fixed-size Memory Block

[C Language Interface]

```
ER ercd = tk_rel_mpf ( ID mpfid, VP blf ) ;
```

[Parameters]

ID	mpfid	MemoryPoolID	Fixed-size memory pool ID
VP	blf	BlockStartAddress	Memory block start address

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mpfid is invalid or cannot be used)
E_NOEXS	Object does not exist (the fixed-size memory pool specified in mpfid does not exist)
E_PAR	Parameter error (blf is invalid, or block returned to wrong memory pool)
E_DACV	Access Protection Violation

[Description]

Returns the memory block specified in blf to the fixed-size memory pool specified in mpfid.

Executing tk_rel_mpf may enable memory block acquisition by another task waiting to allocate memory from the memory pool specified in mpfid, releasing the WAIT state of that task.

When a memory block is returned to a fixed-size memory pool, it must be the same fixed-size memory pool from which the block was allocated. If an attempt to return a memory block to a different memory pool is detected, error code E_PAR is returned. Whether this error detection is permitted or not is implementation-dependent.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified memory pool cannot be accessed due to access protection, E_DACV is returned.

Reference Fixed-size Memory Pool Status tk_ref_mpf

tk_ref_mpf: Refer Fixed-size MemoryPool Status

[C Language Interface]

```
ER ercd = tk_ref_mpf ( ID mpfid, T_RMPF *pk_rmpf ) ;
```

[Parameters]

ID	mpfid	MemoryPoolID	Fixed-size memory pool ID
T_RMPF*	pk_rmpf	Packet to Refer MemoryPool	Address of packet for returning status information

[Return Parameters]

ER	ercd	ErrorCode	Error code

pk_rmpf detail:

VP	exinf	ExtendedInformation	Extended information
ID	wtsk	WaitTaskInformation	Waiting task information
INT	frbcnt	FreeBlockCount	Free block count

——(Other implementation-dependent parameters may be added beyond this point.)——

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mpfid is invalid or cannot be used)
E_NOEXS	Object does not exist (the fixed-size memory pool specified in mpfid does not exist)
E_PAR	Parameter error (the return parameter packet address cannot be used)
E_DACV	Access Protection Violation

[Description]

References the status of the fixed-size memory pool specified in mpfid, passing in return parameters the current free block count frbcnt, waiting task information (wtsk), and extended information (exinf).

wtsk indicates the ID of a task waiting for memory block allocation from this fixed-size memory pool. If multiple tasks are waiting for the fixed-size memory pool, the ID of the task at the head of the queue is returned. If there are no waiting tasks, wtsk = 0 is returned.

If the fixed-size memory pool specified with tk_ref_mpf does not exist, error code E_NOEXS is returned.

At least one of frbcnt = 0 and wtsk = 0 is always true for this system call.

Access protection is applied to this system call.

[Additional Notes]

Whereas frsz returned by tk_ref_mpl gives the total free memory size in bytes, frbcnt returns the number of unused memory blocks.

Although the amount of free memory can be confirmed by this call, whether or not it can be used from the invoking task is not guaranteed. Depending on the protection level of the memory pool, there are cases where access to the acquired memory block is not possible (exception occurs).

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified memory pool cannot be accessed due to access protection, E_DACV is returned.

4.6.2 Variable-size Memory Pool

A variable-size memory pool is an object for dynamically managing memory blocks of any size. Functions are provided for creating and deleting a variable-size memory pool, allocating and returning memory blocks in a variable-size memory pool, and referencing the status of a variable-size memory pool. A variable-size memory pool is an object identified by an ID number called a variable-size memory pool ID.

A variable-size memory pool has a memory space used as the variable-size memory pool (called a variable-size memory pool area or simply memory pool area), and a queue for tasks waiting for memory block allocation. A task wanting to allocate a memory block from a variable-size memory pool that lacks sufficient available memory space goes to WAIT state for variable-size memory block until memory blocks are returned to the pool. A task in this state is put in the task queue of the variable-size memory pool.

In SMP T-Kernel, access protection is applied to system calls that specify a Variable Length Memory Pool ID.

Service calls with different specifications from the T-Kernel 1.00 Specification are summarized in the table below. For more details refer to the explanation for each system call.

Call Name	Function	Different from T-Kernel 1.00 Specification
tk_cre_mpl	Create Variable-size Memory Pool	×
tk_del_mpl	Delete Variable-size Memory Pool	△
tk_get_mpl	Get Variable-size Memory Block	△
tk_rel_mpl	Release Variable-size Memory Block	△
tk_ref_mpl	Reference Variable-size Memory Pool Status	△

Different from T-Kernel 1.00 Specification ○:No ×:Yes △: Different only in that E_DACV is returned due to the access protection

[Additional Notes]

When tasks are waiting for memory block allocation from a variable-size memory pool, they are served in queued order. If, for example, Task A requesting a 400-byte memory block from a variable-size memory pool is queued along with Task B requesting a 100-byte block, in A-B order, then even if 200 bytes of space are free, Task B is made to wait until Task A has acquired the requested memory block.

Create Variable-size Memory Pool tk_cre_mpl

tk_cre_mpl: Create Variable-size MemoryPool

[C Language Interface]

```
ID mplid = tk_cre_mpl ( T_CMPL *pk_cmpl ) ;
```

[Parameters]

T_CMPL* pk_cmpl Packet to Create MemoryPool Information about the variable-size memory pool to be created

pk_cmpl detail:

VP	exinf	ExtendedInformation	Extended information
ATR	mplatr	MemoryPoolAttribute	Memory pool attributes
INT	mplsz	MemoryPoolSize	Memory pool size (in bytes)
ID	domid	DomainID	Domain ID
UB	oname[8]	Object name	DS Object name

——(Other implementation-dependent parameters may be added beyond this point.)——

[Return Parameters]

ID	mplid	MemoryPoolID	Variable-size memory pool ID
	or	ErrorCode	Error Code

[Error Codes]

E_NOMEM	Insufficient memory (memory for control block or memory pool area cannot be allocated)
E_LIMIT	Number of variable-size memory pools exceeds the system limit
E_RSATR	Reserved attribute (mplatr is invalid or cannot be used)
E_PAR	Parameter error (pk_cmpl is invalid, or mplsz is negative or invalid)
E_ID	Invalid ID number (domid is invalid or cannot be used)
E_NOEXS	Object does not exist (domain of domid does not exist)
E_NSPT	Unsupported (Does not have shared memory between processors)
E_ONAME	Specified object name has already been used

[Description]

Creates a variable-size memory pool, assigning it to a variable-size memory pool ID. This system call allocates a memory space for use as a memory pool, based on the information in parameter mplsz, and allocates a control block to the created memory pool.

exinf can be used freely by the user to store miscellaneous information about the created memory pool. The information set in this parameter can be referenced by tk_ref_mpl. If a larger area is needed for indicating user information, or if the information needs to be changed after the memory pool is created, this can be done by allocating separate memory for this purpose and putting the memory packet address in exinf. The OS pays no attention to the contents of exinf.

mplatr indicates system attributes in its low bits and implementation-dependent information in the high bits. The system attributes part of mplatr is as follows.

```
mplatr:= (TA_TFIFO || TA_TPRI) | [TA_ONAME] | [TA_NODISWAI]
         | (TA_RNG0 || TA_RNG1 || TA_RNG2 || TA_RNG3)
         | [TA_DOMID] | [(TA_PROTECTED || TA_PRIVATE || TA_PUBLIC)]
```

TA_TFIFO	Tasks waiting for memory allocation are queued in FIFO order
TA_TPRI	Tasks waiting for memory allocation are queued in priority order
TA_RNGn	Memory access privilege is set to protection level n
TA_ONAME	Specifies DS Object name
TA_NODISWAI	Wait disabling by tk_dis_wai is prohibited
TA_DOMID	Specifies the domain to which the task belongs
TA_PROTECTED	Sets the access protection attribute to protect
TA_PRIVATE	Sets the access protection attribute is set to private
TA_PUBLIC	Sets the access protection attribute is set to public

The queuing order of tasks waiting to acquire memory from a memory pool can be specified in TA_TFIFO or TA_TPRI. If the attribute is TA_TFIFO, tasks are ordered by FIFO, whereas TA_TPRI specifies queuing of tasks in order of their priority setting.

When tasks are queued waiting for memory allocation, memory is allocated in the order of queuing. Even if other tasks in the queue are requesting smaller amounts of memory than the task at the head of the queue, they do not acquire memory blocks before the first task. If, for example, Task A requesting a 400-byte memory block from a variable-size memory pool is queued along with Task B requesting a 100-byte block, in A-B order, then even if 200 bytes of space are free, Task B is made to wait until Task A has acquired the requested memory block.

TA_RNGn is specified to limit the protection levels at which memory can be accessed. Only tasks running at the same or higher protection level than the one specified can access the allocated memory. If a task running at a lower protection level attempts access, a CPU protection fault exception is raised. For example, memory allocated from a memory pool specified as TA_RNG1 can be accessed by tasks running at levels TA_RNG0 or TA_RNG1, but not by tasks running at levels TA_RNG2 or TA_RNG3.

The created memory pool is in resident memory in system space. There is no T-Kernel function for creating a memory pool in task space.

When TA_ONAME is specified, oname is valid and is set as the object name. When TA_ONAME is not specified, the object name is not set. The object name must be unique within the domain to which the memory pool belongs. When an object name that has already been used with another memory pool is specified, E_ONAME is returned. When the length of the character string specified for oname is 0 (initial character is termination 0), the object name is not set regardless of the specification of TA_ONAME.

When TA_DOMID is specified, domid is valid, and the domain of domid is set as the domain to which it belongs. When TA_DOMID is not specified, domid is ignored and is the domain to which the kernel domain belongs. TA_PROTECTED, TA_PRIVATE, and TA_PUBLIC specify the access protection attribute of the memory pool. When either of the access protection attributes is not specified, the access protection is set to the public attribute. In the combination of the domain to which the task belongs and the access protection attribute, memory pools that invoking tasks cannot access due to access protection cannot be created. When the corresponding specification is done, E_PAR is returned.

```
#define TA_TFIFO        0x00000000   /* manage task queue by FIFO */
#define TA_TPRI         0x00000001   /* manage task queue by priority */
#define TA_ONAME        0x00000040   /* DS Object name */
#define TA_NODISWAI     0x00000080   /* reject wait disabling */
#define TA_RNG0         0x00000000   /* protection level 0 */
#define TA_RNG1         0x00000100   /* protection level 1 */
#define TA_RNG2         0x00000200   /* protection level 2 */
#define TA_RNG3         0x00000300   /* protection level 3 */
#define TA_DOMID        0x00010000   /* specifies the domain */
#define TA_PRIVATE      0x00040000   /* sets the protection attribute to private */
#define TA_PROTECTED    0x00080000   /* sets the protection attribute to protect */
#define TA_PUBLIC       0x00000000   /* sets the protection attribute to public*/
```

[Additional Notes]

If the task at the head of the queue waiting for memory allocation has its WAIT state forcibly released or if a different task becomes the first in the queue as a result of a change in task priority, allocation of memory to that task is attempted. If memory can be allocated, the WAIT state of that task is released. In this way, it is possible under some circumstances for memory allocation to take place and task WAIT state to be released even when memory is not released by tk_rel_mpl.

For the sake of portability, the TA_RNGn attribute must be accepted even by a system without an MMU. It is possible, for example, to treat all TA_RNGn as equivalent to TA_RNG0; but no error must be returned.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- TA_DOMID, TA_PROTECTED, TA_PRIVATE, and TA_PUBLIC were added to the memory pool attribute, and the domain to which it belongs and the access protection attribute are specifiable.
- The DS Object name was abolished, and replaced by the establishment of the object name. While the former was a name for debugging, the latter is a name which can be used in general for searching domain ID's, etc. The object name

cannot use the same name with the same type of object in the same domain.

[Rationale for the Specification]

The capability of creating multiple memory pools can be used for memory allocation as needed for error handling or in emergencies, etc.

Delete Variable-size Memory Pool tk_del_mpl

tk_del_mpl: Delete Variable-size MemoryPool

[C Language Interface]

```
ER ercd = tk_del_mpl ( ID mplid ) ;
```

[Parameters]

ID	mplid	MemoryPoolID	Variable-size memory pool ID

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mplid is invalid or cannot be used)
E_NOEXS	Object does not exist (the variable-size memory pool specified in mplid does not exist)
E_DACV	Access Protection Violation

[Description]

Deletes the variable-size memory pool specified in mplid.

No check or notification is made as to whether there are tasks using memory allocated from this memory pool. The system call completes normally even if some blocks have not been returned to the pool.

Issuing this system call releases the memory pool ID number, the control block memory space and the memory pool space itself.

This system call completes normally even if there are tasks waiting for memory block allocation from the deleted memory pool, but error code E_DLT is returned to the tasks in WAIT state.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified memory pool cannot be accessed due to access protection, E_DACV is returned.

tk_get_mpl: Get Variable-size Memory Block

[C Language Interface]

```
ER ercd = tk_get_mpl ( ID mplid, INT blksz, VP *p_blk, TMO tmout ) ;
```

[Parameters]

ID	mplid	MemoryPoolID	Variable-size memory pool ID
INT	blksz	MemoryBlockSize	Memory block size (in bytes)
TMO	tmout	Timeout	Timeout

[Return Parameters]

ER	ercd	ErrorCode	Error code
VP	blk	BlockStartAddress	Block start address

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mplid is invalid or cannot be used)
E_NOEXS	Object does not exist (the variable-size memory pool specified in mplid does not exist)
E_PAR	Parameter error (tmout <= (-2))
E_DLT	The object being waited for was deleted (the memory pool was deleted while waiting)
E_RLWAI	Wait state released (tk_rel_wai received in wait state)
E_DISWAI	Wait released by wait disabled state
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion or in dispatch disabled state)
E_DACV	Access Protection Violation

[Description]

Gets a memory block of size blksz (bytes) from the variable-size memory pool specified in mplid. The start address of the allocated memory block is returned in blk.

The allocated memory is not cleared to zero, and the memory block contents are indeterminate.

If memory cannot be allocated, the task issuing this system call enters WAIT state.

A maximum wait time (timeout) can be set in tmout. If the tmout time elapses before the wait release condition is met (memory space does not become available), the system call terminates, returning timeout error code E_TMOUT.

Only positive values can be set in tmout. The time base for tmout (time unit) is the same as that for system time (= 1 ms).

When TMO_POL = 0 is set in tmout, this means 0 was specified as the timeout value, and E_TMOUT is returned without entering WAIT state even if memory cannot be allocated.

When TMO_FEVR = (-1) is set in tmout, this means infinity was specified as the timeout value, and the task continues to wait for memory allocation without timing out.

The queuing order of tasks waiting for memory block allocation is either FIFO or task priority order, depending on the memory pool attribute.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified memory pool cannot be accessed due to access protection, E_DACV is returned.

tk_rel_mpl: Release Variable-size Memory Block

[C Language Interface]

```
ER ercd = tk_rel_mpl ( ID mplid, VP blk ) ;
```

[Parameters]

ID	mplid	MemoryPoolID	Variable-size memory pool ID
VP	blk	BlockStartAddress	Memory block start address

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mplid is invalid or cannot be used)
E_NOEXS	Object does not exist (the variable-size memory pool specified in mplid does not exist)
E_PAR	Parameter error (blk is invalid, or block returned to wrong memory pool)
E_DACV	Access Protection Violation

[Description]

Returns the memory block specified in blk to the variable-size memory pool specified in mplid.

Executing tk_rel_mpl may enable memory block acquisition by another task waiting to allocate memory from the memory pool specified in mplid, releasing the WAIT state of that task.

When a memory block is returned to a variable-size memory pool, it must be the same variable-size memory pool from which the block was allocated. If an attempt to return a memory block to a different memory pool is detected, error code E_PAR is returned. Whether or not this error detection is carried out is implementation-dependent.

Access protection is applied to this system call.

[Additional Notes]

When memory is returned to a variable-size memory pool in which multiple tasks are queued, multiple tasks may be released at the same time depending on the amount of memory returned and the requested memory size. The task precedence among tasks of the same priority after their WAIT state is released in such a case is the order in which they were queued.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified memory pool cannot be accessed due to access protection, E_DACV is returned.

Reference Variable-size Memory Pool Status tk_ref_mpl

tk_ref_mpl: Refer Variable-size MemoryPool Status

[C Language Interface]

ER ercd = tk_ref_mpl (ID mplid, T_RMPL *pk_rmpl) ;

[Parameters]

ID	mplid	MemoryPoolID	Variable-size memory pool ID
T_RMPL*	pk_rmpl	Packet to Refer MemoryPool	Address of packet for returning status information

[Return Parameters]

ER	ercd	ErrorCode	Error code

pk_rmpl detail:

VP	exinf	ExtendedInformation	Extended information
ID	wtsk	WaitTaskInformation	Waiting task information
INT	frsz	FreeMemorySize	Free memory size (in bytes)
INT	maxsz	MaxMemorySize	Maximum memory space size (in bytes)

──(Other implementation-dependent parameters may be added beyond this point.)──

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mplid is invalid or cannot be used)
E_NOEXS	Object does not exist (the variable-size memory pool specified in mplid does not exist)
E_PAR	Parameter error (the address of the return parameter packet cannot be used)
E_DACV	Access Protection Violation

[Description]

References the status of the variable-size memory pool specified in mplid, passing in return parameters the size of total free space (frsz), the maximum size of memory immediately available (maxsz), waiting task information (wtsk), and extended information (exinf).

wtsk indicates the ID of a task waiting for memory block allocation from this variable-size memory pool. If multiple tasks are waiting for the variable-size memory pool, the ID of the task at the head of the queue is returned. If there are no waiting tasks, wtsk = 0 is returned.

. If the variable-size memory pool specified with tk_ref_mpl does not exist, error code E_NOEXS is returned.

Access protection is applied to this system call.

[Additional Notes]

While the total size of the free memory in frsz of tk_ref_mpl is returned by the number of bytes, the unused memory block count is returned in frbcnt of tk_ref_mpf.

Although the amount of free memory can be confirmed by this call, whether or not it can be used from the invoking task is not guaranteed. Depending on the protection level of the memory pool, there may be cases where the acquired memory block cannot be accessed (exception occurs).

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified memory pool cannot be accessed due to access protection, E_DACV is returned.

4.7 Time Management Functions

Time management functions are for performing time-dependent processing. They include functions for system time management, cyclic handlers, and alarm handlers.

The general name used here for cyclic handlers and alarm handlers is time event handlers.

4.7.1 System Time Management

System time management functions are for manipulating system time. Functions are provided for setting and reading system clock and for reading system operating time.

There are no differences between the system time management of SMP T-Kernel and the T-Kernel 1.00 Specification. System calls for system time management are summarized in the table below. For more details refer to the explanation for each system call.

Call Name	Function	Different from T-Kernel 1.00 Specification
tk_set_tim	Set Time	○
tk_get_tim	Get System Time	○
tk_get_otm	Get Operating Time	○

Different from T-Kernel 1.00 Specification ○:No ×:Yes △: Different only in that E_DACV is returned due to the access protection

Set Time tk_set_tim

tk_set_tim: Set Time

[C Language Interface]

```
ER ercd = tk_set_tim ( SYSTIM *pk_tim ) ;
```

[Parameters]

SYSTIM* pk_tim	Packet of CurrentTime	Address of current time packet

pk_tim detail

SYSTIM	systim	CurrentSystemTime	Current system time

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_PAR	Parameter error (pk_tim is invalid, or time setting is invalid)

[Description]

Sets the system clock to the value specified in systim.

System time is expressed as cumulative milliseconds from 0:00:00 (GMT), January 1, 1985.

[Additional Notes]

The relative time specified in RELTIM or TMO does not change even if the system clock is changed by calling tk_set_tim during system operation. For example, if a timeout is set to elapse in 60 seconds and the system clock is advanced by 60 seconds by tk_set_tim while waiting for the timeout, the timeout occurs not immediately but 60 seconds after it was set. Instead, tk_set_tim changes the system time at which the timeout occurs.

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

tk_get_tim: Get Time

[C Language Interface]

```
ER ercd = tk_get_tim ( SYSTIM *pk_tim ) ;
```

[Parameters]

SYSTIM*	pk_tim	Packet of CurrentTime	Address of time packet

[Return Parameters]

ER	ercd	ErrorCode	Error code

pk_tim detail

SYSTIM	systim	CurrentSystemTime	Current system time

[Error Codes]

E_OK	Normal completion
E_PAR	Parameter error (pk_tim is invalid)

[Description]
Reads the current value of the system clock and returns it in systim.
System time is expressed as cumulative milliseconds from 0:00:00 (GMT), January 1, 1985.

[Items Concerning SMP T-Kernel]
There are no differences from the T-Kernel 1.00 Specification.

tk_get_otm: Get Operating Time

[C Language Interface]

```
ER ercd = tk_get_otm ( SYSTIM *pk_tim ) ;
```

[Parameters]

SYSTIM*	pk_tim	Address of packet for returning operating time

[Return Parameters]

ER	ercd	Error code

pk_tim detail:

SYSTIM	opetim	System operating time

[Error Codes]

E_OK	Normal completion
E_PAR	Parameter error (pk_tim is invalid)

[Description]

Gets the system operating time (up time).

System operating time, unlike system time, indicates the length of time elapsed linearly since the system was started. It is not affected by clock settings made by tk_set_tim.

System operating time must have the same precision as system time.

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

4.7.2 Cyclic Handler

A cyclic handler is a time event handler started at regular intervals. Cyclic handler functions are provided for creating and deleting a cyclic handler, activating and deactivating a cyclic handler operation, and referencing cyclic handler status. A cyclic handler is an object identified by an ID number called a cyclic handler ID.

The time interval at which a cyclic handler is started (cycle time) and the cycle phase are specified for each cyclic handler when it is created. When a cyclic handler operation is requested, T-Kernel determines the time at which the cyclic handler should not be started next based on the cycle time and cycle phase set for it. When a cyclic handler is created, the time when it is to be started next is the time of its creation plus the cycle phase. When the time comes to start a cyclic handler, exinf, containing extended information about the cyclic handler, is passed to it as a starting parameter. The time when the cyclic handler is started plus its cycle time becomes the next start time. Sometimes when a cyclic handler is activated, the next start time will be newly set.

In principle the cycle phase of a cyclic handler is no longer than its cycle time. The behavior if the cycle phase is made longer than the cycle time is implementation-dependent.

A cyclic handler has two activation states, active and inactive. While a cyclic handler is inactive, it is not started even when its start time arrives, although calculation of the next start time does take place. When a system call for activating a cyclic handler is called (tk_sta_cyc), the cyclic handler goes to active state, and the next start time is decided if necessary. When a system call for deactivating a cyclic handler is called (tk_stp_cyc), the cyclic handler goes to inactive state. Whether a cyclic handler upon creation is active or inactive is decided by a cyclic handler attribute.

The cycle phase of the cyclic handler is a relative time specifying the first time the cyclic handler is to be started, in relation to the time when the system call creating it was invoked. The cycle time of a cyclic handler is likewise a relative time, specifying the next time the cyclic handler is to be started in relation to the time it should have started (not the time it started). For this reason, the intervals between times the cyclic handler is started will individually be shorter than the cycle time in some cases, but their average over a longer time span will match the cycle time.

In SMP T-Kernel, access protection is applied to system calls that specify a Cyclic Handler ID.

Differences in system calls for cyclic handlers with the T-Kernel 1.00 Specification are summarized in the table below. For more details refer to the explanation for each system call.

Call Name	Function	Different from T-Kernel 1.00 Specification
tk_cre_cyc	Create Cyclic Handler	×
tk_del_cyc	Delete Cyclic Handler	△
tk_sta_cyc	Start Cyclic Handler	△
tk_stp_cyc	Stop Cyclic Handler	△
tk_ref_cyc	Reference Cyclic Handler Status	△

Different from T-Kernel 1.00 Specification ○:No ×:Yes △: Different only in that E_DACV is returned due to the access protection

tk_cre_cyc:Create Cyclic Handler

[C Language Interface]

ID cycid = tk_cre_cyc (T_CCYC *pk_ccyc) ;

[Parameters]

T_CCYC* pk_ccyc Packet to Define CyclicHandler Address of cyclic handler definition packet

pk_ccyc detail:

VP	exinf	ExtendedInformation	Extended information
ATR	cycatr	CyclicHandlerAttribute	Cyclic handler attributes
FP	cychdr	CyclicHandlerAddress	Cyclic handler address
RELTIM	cyctim	CycleTime	Cycle time
RELTIM	cycphs		Cycle phase
ID	domid	DomainID	Domain ID
UW	assprc	AssignProcessor	Execution processor specification
UB	oname[8]	Object name	DS Object name

——(Other implementation-dependent parameters may be added beyond this point.)——

[Return Parameters]

ID	cycid	CyclicHandlerID	Cyclic handler ID
	or	ErrorCode	Error Code

[Error Codes]

E_NOMEM	Insufficient memory (memory for control block cannot be allocated)
E_LIMIT	Number of cyclic handlers exceeds the system limit
E_RSATR	Reserved attribute (cycatr is invalid or cannot be used)
E_PAR	Parameter error (pk_ccyc, cychdr, cyctim, cycphs or assprc is invalid or cannot be used)
E_ID	Invalid ID number (domid is invalid or cannot be used)
E_NOEXS	Object does not exist (domain of domid does not exist)
E_ONAME	Specified object name has already been used

[Description]

Creates a cyclic handler, assigning it to a cyclic handler ID. A cyclic handler is a handler running at specified intervals as a task-independent portion.

exinf can be used freely by the user to store miscellaneous information about the created cyclic handler. The information set in this parameter can be referenced by tk_ref_cyc. If a larger area is needed for indicating user information, or if the information needs to be changed after the cyclic handler is created, this can be done by allocating separate memory for this purpose and putting the memory packet address in exinf. The OS pays no attention to the contents of exinf.

cycatr indicates system attributes in its low bits and implementation-dependent information in the high bits. The system attributes part of cycatr is as follows.

cycatr := (TA_ASM ｜｜ TA_HLNG) | [TA_STA] | [TA_PHS] | [TA_ONAME]
| [TA_ASSPRC] | [TA_DOMID] | [(TA_PROTECTED || TA_PRIVATE || TA_PUBLIC)]

TA_ASM	The handler is written in assembly language
TA_HLNG	The handler is written in high-level language
TA_STA	Activate immediately upon cyclic handler creation
TA_PHS	Save the cycle phase
TA_ONAME	Specifies DS Object name
TA_DOMID	Specifies the domain to which the task belongs
TA_ASSPRC	Specifies the execution processor

TA_PROTECTED	Sets the access protection attribute to protect
TA_PRIVATE	Sets the access protection attribute is set to private
TA_PUBLIC	Sets the access protection attribute is set to public

cychdr specifies the cyclic handler start address, cyctim the cycle time, and cycphs the cycle phase.

When the TA_HLNG attribute is specified, the cyclic handler is started via a high-level language support routine. The high-level language support routine takes care of saving and restoring register values. The cyclic handler terminates by a simple return from the function. The cyclic handler takes the following format when the TA_HLNG attribute is specified.

```
void cychdr( VP exinf )
{
        /*
                Processing
        */

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

The cyclic handler format when the TA_ASM attribute is specified is implementation-dependent, but exinf must be passed to the handler in a starting parameter when it starts.

cycphs indicates the length of time until the cyclic hander is initially started after being created by tk_cre_cyc. Thereafter it is started periodically at the interval set in cyctim. If zero is specified for cycphs, the cyclic handler starts immediately after it is created. Zero cannot be specified for cyctim.

The starting of the cyclic handler for the nth time occurs after at least chcphs + chctim × (n - 1) time has elapsed from the cyclic handler creation.

When TA_STA is specified, the cyclic handler goes to active state immediately on creation, and starts at the intervals noted above. If TA_STA is not specified, the cycle time is calculated but the cyclic handler is not actually started.

When TA_PHS is specified, then even if tk_sta_cyc is called activating the cyclic handler, the cycle time is not reset, and the cycle time calculated as above from the time of cyclic handler creation continues to apply. If TA_PHS is not specified, calling tk_sta_cyc resets the cycle time and the cyclic handler is started at cyctim intervals measured from the time tk_sta_cyc was called. Note that the resetting of cycle time by tk_sta_cyc does not affect cycphs. In this case, the starting of the cyclic handler for the nth time occurs after at least cyctim × n has elapsed from the calling of tk_sta_cyc.

When TA_ONAME is specified, oname is valid and is set as the object name. When TA_ONAME is not specified, the object name is not set. The object name must be unique within the domain to which the cyclic handler belongs. When an object name that has already been used with another cyclic handler is specified, E_ONAME is returned. When the length of the character string specified for oname is 0 (initial character is termination 0), the object name is not set regardless of the specification of TA_ONAME.

When TA_ASSPRC is specified, assprc becomes valid and the processor that executes the cyclic handler is set. The value of assprc is defined as follows.

```
#define TP_PRC1      0x00000001    Processor ID Number 1 Processor
#define TP_PRC2      0x00000002    Processor ID Number 2 Processor
#define TP_PRC3      0x00000004    Processor ID Number 3 Processor
#define TP_PRC4      0x00000008    Processor ID Number 4 Processor
                         ～
#define TP_PRC32     0x80000000    Processor ID Number 32 Processor
```

Only one processor can be specified to execute the cyclic handler. Multiple processors cannot be specified. When TA_ASSPRC is not specified, assprc is ignored. In this case, the processor that executes the cyclic handler is implementation-defined.

When TA_DOMID is specified, domid is valid, and the domain of domid is set as the domain to which it belongs. When TA_DOMID is not specified, domid is ignored and is the domain to which the kernel domain belongs.

TA_PROTECTED, TA_PRIVATE, and TA_PUBLIC specify the access protection attribute of the cyclic handler. When either of the access protection attributes is not specified, the access protection is set to the public attribute. In the combination of the domain to which the task belongs and the access protection attribute, cyclic handlers that invoking tasks cannot access due to access protection cannot be created. When the corresponding specification is done, E_PAR is returned.

During execution of the cyclic handler, even if a dispatch becomes necessary, priority is first given to execute the cyclic handler until completion, and dispatch is delayed until completion of the cyclic handler (principle of delay dispatch). However, dispatch is possible in other processors than the processor executing the cyclic handler. Therefore, what is delayed is the dispatch by the processor executing the cyclic handler.

A cyclic handler runs as a task-independent portion. As such, it is not possible to call in a cyclic handler a system call that can enter WAIT state, or one that is intended for the invoking task.

```
#define TA_ASM          0x00000000   /* assembly program */
#define TA_HLNG         0x00000001   /* high-level language program */
#define TA_STA          0x00000002   /* activate cyclic handler */
#define TA_PHS          0x00000004   /* save cyclic handler cycle phase */
#define TA_ONAME        0x00000040   /* DS Object name */
#define TA_DOMID        0x00010000   /* specifies the domain */
#define TA_ASSPRC       0x00020000   /* specifies the execution processor*/
#define TA_PRIVATE      0x00040000   /* sets the protection attribute to private */
#define TA_PROTECTED    0x00080000   /* sets the protection attribute to protect */
#define TA_PUBLIC       0x00000000   /* sets the protection attribute to public*/
```

[Additional Notes]

Once a cyclic handler is defined, it continues to run at the specified cycles either until tk_stp_cyc is called to deactivate it or until it is deleted. There is no parameter to specify the number of cycles in tk_cre_cyc.

When multiple time event handlers or interrupt handlers operate at the same time, it is an implementation-dependent matter whether to have them run serially (after one handler exits, another starts) or nested (one handler operation is suspended, another runs, and when that one finishes the previous one resumes). In either case, since time event handlers and interrupt handlers run as task-independent portions, the principle of delayed dispatching applies.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- TA_DOMID, TA_PROTECTED, TA_PRIVATE, and TA_PUBLIC were added to the cyclic handler attribute, and the domain to which it belongs and the access protection attribute are specifiable.
- TA_ASSPRC was added to the cyclic handler attribute, and the processor that executes the cyclic handler is specifiable.
- The DS Object name was abolished, and replaced by the establishment of the object name. While the former was a name for debugging, the latter is a name which can be used in general for searching domain ID's, etc. The object name cannot use the same name with the same type of object in the same domain.

Delete Cyclic Handler tk_del_cyc

tk_del_cyc: Delete Cyclic Handler

[C Language Interface]

```
ER ercd = tk_del_cyc ( ID cycid ) ;
```

[Parameters]

ID	cycid	CyclicHandlerID	Cyclic handler ID

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (cycid is invalid or cannot be used)
E_NOEXS	Object does not exist (the cyclic handler specified in cycid does not exist)
E_DACV	Access Protection Violation

[Description]

Deletes the cyclic handler specified in cycid.
Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified cyclic handler cannot be accessed due to access protection, E_DACV is returned.

Start Cyclic Handler tk_sta_cyc

tk_sta_cyc: Start Cyclic Handler

[C Language Interface]

ER ercd = tk_sta_cyc (ID cycid) ;

[Parameters]

ID	cycid	CyclicHandlerID	Cyclic handler ID

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (cycid is invalid or cannot be used)
E_NOEXS	Object does not exist (the cyclic handler specified in cycid does not exist)
E_DACV	Access Protection Violation

[Description]

Activates a cyclic handler, putting it in active state.

If the TA_PHS attribute was specified, the cycle time of the cyclic handler is not reset when the cyclic handler goes to active state. If it was already in active state when this system call was executed, it continues unchanged in active state.

If the TA_PHS attribute was not specified, the cycle time is reset when the cyclic handler goes to active state. If it was already in active state, it continues in active state but its cycle time is reset. In this case, the next time the cyclic handler starts is after cyctim has elapsed.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified cyclic handler cannot be accessed due to access protection, E_DACV is returned.

tk_stp_cyc: Stop Cyclic Handler

[C Language Interface]

ER ercd = tk_stp_cyc (ID cycid) ;

[Parameters]

ID	cycid	CyclicHandlerID	Cyclic handler ID

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (cycid is invalid or cannot be used)
E_NOEXS	Object does not exist (the cyclic handler specified in cycid does not exist)
E_DACV	Access Protection Violation

[Description]

Deactivates a cyclic handler, putting it in inactive state. If the cyclic handler was already in inactive state, this system call has no effect (no operation).

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified cyclic handler cannot be accessed due to access protection, E_DACV is returned.

Reference Cyclic Handler Status tk_ref_cyc

tk_ref_cyc: Refer Cyclic Handler Status

[C Language Interface]

```
ER ercd = tk_ref_cyc ( ID cycid, T_RCYC *pk_rcyc ) ;
```

[Parameters]

ID	cycid	CyclicHandlerID	Cyclic handler ID
T_RCYC*	pk_rcyc	Packet to Refer CyclicHandler	Address of packet for returning status information

[Return Parameters]

ER	ercd	ErrorCode	Error code

pk_rcyc detail:

VP	exinf	ExtendedInformation	Extended information
RELTIM	lfttim	LeftTime	Time remaining until the next start time
UINT	cycstat	CyclicHandlerStatus	Cyclic handler activation state

——(Other implementation-dependent parameters may be added beyond this point.)——

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (cycid is invalid or cannot be used)
E_NOEXS	Object does not exist (the cyclic handler specified in cycid does not exist)
E_PAR	Parameter error (the address of the return parameter packet cannot be used)
E_DACV	Access Protection Violation

[Description]

References the status of the cyclic handler specified in cycid, passing in return parameters the cyclic handler activation state cycstat, the time remaining until the next start lfttim, and extended information exinf.

The following information is returned in cycstat.

cycstat:= (TCYC_STP | TCYC_STA)

```
#define TCYC_STP    0x00    /* cyclic handler is inactive */
#define TCYC_STA    0x01    /* cyclic handler is active */
```

If the cyclic handler specified in cycid does not exist, error code E_NOEXS is returned.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified cyclic handler cannot be accessed due to access protection, E_DACV is returned.

4.7.3 Alarm Handler

An alarm handler is a time event handler that starts at a specified time. Functions are provided for creating and deleting an alarm handler, activating and deactivating the alarm handler, and referencing the alarm handler status. An alarm handler is an object identified by an ID number called an alarm handler ID.

The time at which an alarm handler starts (called the alarm time) can be set independently for each alarm handler. When the alarm time arrives, exinf, containing extended information about the alarm handler, is passed to it as a starting parameter.

After an alarm handler is created, initially it has no alarm time set and is in inactive state. The alarm time is set when the alarm handler is activated by calling tk_sta_alm, as relative time from the time that the system call is executed. When tk_stp_alm is called deactivating the alarm handler, the alarm time setting is canceled. Similarly, when the alarm time arrives and the alarm handler runs, the alarm time is canceled and the alarm handler becomes inactive.

In SMP T-Kernel, access protection is applied to system calls that specify an Alarm Handler ID.

System calls which are different than the T-Kernel 1.00 Specification are summarized in the table below. For more details refer to the explanation for each system call.

Call Name	Function	Different from T-Kernel 1.00 Specification
tk_cre_alm	Create Alarm Handler	×
tk_del_alm	Delete Alarm Handler	△
tk_sta_alm	Start Alarm Handler	△
tk_stp_alm	Stop Alarm Handler	△
tk_ref_alm	Reference Alarm Handler Status	△

Different from T-Kernel 1.00 Specification ○:No ×:Yes △: Different only in that E_DACV is returned due to the access protection

tk_cre_alm: Create Alarm Handler

[C Language Interface]

```
ID almid = tk_cre_alm ( T_CALM *pk_calm ) ;
```

[Parameters]

T_CALM* pk_calm Packet to Define AlarmHandler Address of packet for alarm handler definition

pk_calm detail:

VP	exinf	ExtendedInformation	Extended information
ATR	almatr	AlarmHandlerAttribute	Alarm handler attributes
FP	almhdr	AlarmHandlerAddress	Alarm handler address
ID	domid	DomainID	Domain ID
UW	assprc	AssignProcessor	Execution processor specification
UB	oname[8]	Object name	DS Object name

[Return Parameters]

ID	almid	AlarmHandlerID	Alarm handler ID
	or	ErrorCode	Error Code

[Error Codes]

E_NOMEM	Insufficient memory (memory for control block cannot be allocated)
E_LIMIT	Number of alarm handlers exceeds the system limit
E_RSATR	Reserved attribute (almatr is invalid or cannot be used)
E_PAR	Parameter error (almno, pk_calm, almatr, almhdr, or assprc is invalid or cannot be used)
E_ID	Invalid ID number (domid is invalid or cannot be used)
E_NOEXS	Object does not exist (domain of domid does not exist)
E_ONAME	Specified object name has already been used

[Description]

Creates an alarm handler, assigning it to an alarm handler ID. An alarm handler is a handler running at the specified time as a task-independent portion.

exinf can be used freely by the user to store miscellaneous information about the created alarm handler. The information set in this parameter can be referenced by tk_ref_alm. If a larger area is needed for indicating user information, or if the information needs to be changed after the alarm handler is created, this can be done by allocating separate memory for this purpose and putting the memory packet address in exinf. The OS pays no attention to the contents of exinf.

almatr indicates system attributes in its low bits and implementation-dependent information in the high bits. The system attributes part of almatr is as follows.

```
almatr := (TA_ASM || TA_HLNG) | [TA_ONAME]
    | [TA_ASSPRC] | [TA_DOMID] | [(TA_PROTECTED || TA_PRIVATE || TA_PUBLIC)]
```

TA_ASM	The handler is written in assembly language
TA_HLNG	The handler is written in a high-level language
TA_ONAME	Specifies DS Object name
TA_ASSPRC	Specifies the execution processor
TA_DOMID	Specifies the domain to which the task belongs
TA_PROTECTED	Sets the access protection attribute to protect
TA_PRIVATE	Sets the access protection attribute is set to private
TA_PUBLIC	Sets the access protection attribute is set to public

almhdr specifies the alarm handler start address.

When the TA_HLNG attribute is specified, the alarm handler is started via a high-level language support routine. The high-level language support routine takes care of saving and restoring register values. The alarm handler terminates by a simple return from the function. The alarm handler takes the following format when the TA_HLNG attribute is specified.

```
void almhdr( VP exinf )
{
        /*
                Processing
        */

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

The alarm handler format when the TA_ASM attribute is specified is implementation-dependent, but exinf must be passed to the handler in a parameter when it starts.

When TA_ONAME is specified, oname is valid and is set as the object name. When TA_ONAME is not specified, the object name is not set. The object name must be unique within the domain to which the cyclic handler belongs. When an object name that has already been used with another cyclic handler is specified, E_ONAME is returned. When the length of the character string specified for oname is 0 (initial character is termination 0), the object name is not set regardless of the specification of TA_ONAME.

When TA_ASSPRC is specified, assprc becomes valid and the processor that executes the alarm handler is set. The value of assprc is defined as follows.

```
#define TP_PRC1     0x00000001     Processor ID Number 1 Processor
#define TP_PRC2     0x00000002     Processor ID Number 2 Processor
#define TP_PRC3     0x00000004     Processor ID Number 3 Processor
#define TP_PRC4     0x00000008     Processor ID Number 4 Processor
                       ～
#define TP_PRC32    0x80000000     Processor ID Number 32 Processor
```

Only one processor can be specified to execute the alarm handler. Multiple processors cannot be specified. When TA_ASSPRC is not specified, assprc is ignored. In this case, the processor that executes the alarm handler is implementation-defined.

When TA_DOMID is specified, domid is valid, and the domain of domid is set as the domain to which it belongs. When TA_DOMID is not specified, domid is ignored and is the domain to which the kernel domain belongs.

TA_PROTECTED, TA_PRIVATE, and TA_PUBLIC specify the access protection attribute of the alarm handler. When either of the access protection attributes is not specified, the access protection is set to the public attribute. In the combination of the domain to which the task belongs and the access protection attribute, alarm handlers that invoking tasks cannot access due to access protection cannot be created. When the corresponding specification is done, E_PAR is returned.

Completion of execution by the alarm handler has precedence even if dispatching is necessary, dispatching takes place only when the alarm handler terminates. In other words, a dispatch request occurring while an alarm handler is running is not processed immediately, but is delayed until the alarm handler terminates. This is called delayed dispatching. However, dispatch is possible in other processors than the processor executing the alarm handler. Therefore, what is delayed is the dispatch by the processor executing the alarm handler.

An alarm handler runs as a task-independent portion. As such, it is not possible to call in an alarm handler a system call that can enter WAIT state, or one that is intended for the invoking task.

```
#define TA_ASM          0x00000000     /* Indicates that the task is written in assembly language */
#define TA_HLNG         0x00000001     /* Indicates that the task is written in high-level language */
#define TA_ONAME        0x00000040     /* Specifies the object name */
#define TA_DOMID        0x00010000     /* specifies the domain */
#define TA_ASSPRC       0x00020000     /* specifies the execution processor*/
#define TA_PRIVATE      0x00040000     /* sets the protection attribute to private */
#define TA_PROTECTED    0x00080000     /* sets the protection attribute to protect */
#define TA_PUBLIC       0x00000000     /* sets the protection attribute to public*/
```

[Additional Notes]

When multiple time event handlers or interrupt handlers operate at the same time, it is an implementation-dependent matter whether to have them run serially (after one handler exits, another starts) or nested (one handler operation is suspended, another runs, and when that one finishes the previous one resumes). In either case, since time event handlers and interrupt handlers run as task-independent portions, the principle of delayed dispatching applies.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- TA_DOMID, TA_PROTECTED, TA_PRIVATE, and TA_PUBLIC were added to the alarm handler attribute, and the domain to which it belongs and the access protection attribute are specifiable.
- TA_ASSPRC was added to the alarm handler attribute, and the processor that executes the alarm handler is specifiable.
- The DS Object name was abolished, and replaced by the establishment of the object name. While the former was a name for debugging, the latter is a name which can be used in general for searching domain ID's, etc. The object name cannot use the same name with the same type of object in the same domain.

tk_del_alm: Delete Alarm Handler

[C Language Interface]

```
ER ercd = tk_del_alm ( ID almid ) ;
```

[Parameters]

ID	almid	AlarmHandlerID	Alarm handler ID

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (almid is invalid or cannot be used)
E_NOEXS	Object does not exist (the alarm handler specified in almid does not exist)
E_DACV	Access Protection Violation

[Description]

Deletes the alarm handler specified in almid.
Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified alarm handler cannot be accessed due to access protection, E_DACV is returned.

tk_sta_alm: Start Alarm Handler

[C Language Interface]

```
ER ercd = tk_sta_alm ( ID almid, RELTIM almtim ) ;
```

[Parameters]

ID	almid	AlarmHandlerID	Alarm handler ID
RELTIM	almtim		Alarm handler start time (alarm time)

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (almid is invalid or cannot be used)
E_NOEXS	Object does not exist (the alarm handler specified in almid does not exist)
E_DACV	Access Protection Violation

[Description]

Sets the alarm time of the alarm handler specified in almid to the time given in almtim, putting the alarm handler in active state. almtim is specified as relative time from the time of calling tk_sta_alm. After the time specified in almtim has elapsed, the alarm handler starts. If the alarm handler is already active when this system call is invoked, the existing almtim setting is canceled and the alarm handler is activated again with the alarm time specified here.

If almtim = 0 is set, the alarm handler starts as soon as it is activated.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified alarm handler cannot be accessed due to access protection, E_DACV is returned.

tk_stp_alm: Stop Alarm Handler

[C Language Interface]

```
ER ercd = tk_stp_alm ( ID almid ) ;
```

[Parameters]

ID	almid	AlarmHandlerID	Alarm handler ID

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (almid is invalid or cannot be used)
E_NOEXS	Object does not exist (the alarm handler specified in almid does not exist)
E_DACV	Access Protection Violation

[Description]

Cancels the alarm time of the alarm handler specified in almid, putting it in inactive state. If it was already in the inactive state, this system call has no effect (no operation).

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified alarm handler cannot be accessed due to access protection, E_DACV is returned.

Reference Alarm Handler Status tk_ref_alm

tk_ref_alm: Refer Alarm Handler Status

[C Language Interface]

```
ER ercd = tk_ref_alm ( ID almid, T_RALM *pk_ralm ) ;
```

[Parameters]

ID	almid	AlarmHandlerID	Alarm handler ID
T_RALM*	pk_ralm	Packet to Refer AlarmHandler	Address of packet for returning status information

[Return Parameters]

ER	ercd	ErrorCode	Error code

pk_ralm detail:

VP	exinf	ExtendedInformation	Extended information
RELTIM	lfttim	LeftTime	Time remaining until the handler starts
UINT	almstat		Alarm handler activation state

——(Other implementation-dependent parameters may be added beyond this point.)——

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (almid is invalid or cannot be used)
E_NOEXS	Object does not exist (the alarm handler specified in almid does not exist)
E_PAR	Parameter error (the address of the return parameter packet cannot be used)
E_DACV	Access Protection Violation

[Description]

References the status of the alarm handler specified in almno, passing in return parameters the time remaining until the handler starts (lfttim), and extended information (exinf).

The following information is returned in almstat.

almstat:= (TALM_STP | TALM_STA)

```
#define TALM_STP    0x00    /* The alarm handler is inactive */
#define TALM_STA    0x01    /* The alarm handler is active */
```

If the alarm handler is active (TALM_STA), lfttim returns the relative time until the alarm handler is scheduled to start. This value is within the range almtim >= lfttim >= 0 specified with tk_sta_alm. Since lfttim is decremented with each timer interrupt, lfttim = 0 means the alarm handler will start at the next timer interrupt.

If the alarm handler is inactive (TALM_STP), lfttim is indeterminate.

If the alarm handler specified with tk_ref_alm in almid does not exist, error code E_NOEXS is returned.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- When the specified alarm handler cannot be accessed due to access protection, E_DACV is returned.

4.8 Domain Management Functions

Domains are objects for managing the location of kernel objects. The domain management function includes a function to create or delete domains, a function to search for objects belonging to a specific domain by using the name, and a function to acquire domain related information on each object. Domains are objects which can be identified by ID numbers. The ID number of a domain is in turn called the domain ID.

Domains wait for the management table that registers kernel objects. The registration and deletion of an object on the management table is conducted when the function of the domain is called within the system call when the object is created or deleted. The registered object can search for the ID number from the object name.

When the kernel object operates by a system call, access protection is conducted based on the domain to which the target object belongs, access protection attribute, and domain to which the program that issued the system call (tasks, handlers) belongs.

System calls for domain management functions are summarized in the table below. For more details refer to the explanation for each system call.

Call Name	Function	Different from T-Kernel 1.00 Specification
tk_cre_dom	Create domain	※
tk_del_dom	Delete domain	※
tk_fnd_xxx	Object ID retrieval	※
tk_dmi_xxx	Get domain information	※
tk_get_kdm	Get kernel domain ID	※
tk_ref_dom	Get domain status	※

*Since it is a new system call, it does not exist in the T-Kernel 1.00 specification.

tk_cre_dom: Create Domain

[C Language Interface]

ID domid = tk_cre_dom (T_CDOM *pk_cdom) ;

[Parameters]

T_CDOM* pk_cdom Packet to Create Domain Information about the domain to be created

pk_cdom details

VP	exinf	ExtendedInformation	Extended information
ATR	domatr	DomainAttribute	Domain attribute
ID	domid	DomainID	Domain ID
UB	oname[8]	Object name	Object name

——(Other implementation-dependent parameters may be added beyond this point.)——

[Return Parameters]

ID	domid	DomainID	Domain ID
	or	ErrorCode	Error code

[Error Codes]

E_NOMEM	Insufficient memory (memory for control block cannot be allocated)
E_LIMIT	Domain count exceeds the system limit
E_RSATR	Reserved attribute (domatr is invalid or cannot be used)
E_PAR	Parameter error (pk_cdom is invalid)
E_ID	Invalid ID number (domid is invalid or cannot be used)
E_NOEXS	Object does not exist (domain of domid does not exist)
E_ONAME	Specified object name has already been used

[Description]

Creates a domain, assigning to it a domain ID. This system call allocates a control block to the created domain and performed initialization.

exinf can be used freely by the user to set miscellaneous information about the created domain. The information set in this parameter can be referenced by tk_ref_dom. If a larger area is needed for indicating user information, or if the information may need to be changed after the domain is created, this can be done by allocating separate memory for this purpose and putting the memory packet address in exinf. The kernel pays no attention to the contents of exinf.

domatr indicates system attributes in its low bits and implementation-dependent information in the high bits. The system attributes part of tskatr is as follows.

domatr := [TA_ONAME] | [TA_DOMID]

TA_ONAME	Specifies the object name
TA_DOMID	Specifies the domain to which it belongs

When TA_ONAME is specified, oname is valid and is set as the object name. When TA_ONAME is not specified, the object name is not set. The object name must be unique within the domain to which the domain belongs. When an object name that has already been used with another domain is specified, E_ONAME is returned. When the length of the character string specified for oname is 0 (initial character is termination 0), the object name is not set regardless of the specification of TA_ONAME.

When TA_DOMID is specified, domid is valid, and the domain of domid is set as the domain to which it belongs. When

TA_DOMID is not specified, domid is ignored and is the domain to which the kernel domain belongs. All domains are public attributes and access protection attributes cannot be specified.

```
#define TA_ONAME     0x00000040    /* object name */
#define TA_DOMID     0x00010000    /*specify domain*/
```

[Items Concerning SMP T-Kernel]

This call does not exist in the T-Kernel 1.00 Specification.

Delete Domain tk_del_dom

tk_del_dom: Delete Domain

[C Language Interface]

```
ER ercd = tk_del_dom ( ID domid ) ;
```

[Parameters]

ID	domid	DomainID	Domain ID

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (domid is invalid or cannot be used)
E_NOEXS	Object does not exist (domain of domid does not exist)
E_OBJ	Object status invalid (object belonging to the domain exists)

[Description]

Deletes the domain specified in domid.
The domain ID and control block area are released as a result of this system call.
When an object that belongs to the target domain exists, the domain is not deleted and E_OBJ is returned.
The kernel domain cannot be deleted. When it is to be deleted, E_OBJ is returned.

[Items Concerning SMP T-Kernel]

This call does not exist in the T-Kernel 1.00 Specification.

tk_fnd_dom, tk_fnd_tsk, tk_fnd_sem,tk_fnd_flg, tk_fnd_mbx, tk_fnd_mtx, tk_fnd_mbf, tk_fnd_por, tk_fnd_mpf, tk_fnd_mpl, tk_fnd_alm, tk_fnd_cyc

ID Retrieval of Each Object

tk_fnd_xxx: Find Object ID

[C Language Interface]

```
ID domid = tk_fnd_dom ( ID domid, UB *oname ) ;     /* Domain */
ID tskid = tk_fnd_tsk ( ID domid, UB *oname ) ;     /* Task */
ID semid = tk_fnd_sem ( ID domid, UB *oname ) ;     /* Semaphore */
ID flgid = tk_fnd_flg ( ID domid, UB *oname ) ;     /* Event Flag */
ID mbxid = tk_fnd_mbx ( ID domid, UB *oname ) ;     /* Mailbox */
ID mtxid = tk_fnd_mtx ( ID domid, UB *oname ) ;     /* Mutex */
ID mbfid = tk_fnd_mbf ( ID domid, UB *oname ) ;     /* Message Buffer */
ID porid = tk_fnd_por ( ID domid, UB *oname ) ;     /* Rendezvous Port */
ID mpfid = tk_fnd_mpf ( ID domid, UB *oname ) ;     /* Fixed-size Memory Pool */
ID mplid = tk_fnd_mpl ( ID domid, UB *oname ) ;     /* Variable Length Memory Pool */
ID almid = tk_fnd_alm ( ID domid, UB *oname ) ;     /* Alarm Handler */
ID cycid = tk_fnd_cyc ( ID domid, UB *oname ) ;     /* Cyclic Handler */
```

[Parameters]

ID	domid	Domain ID
UB*	oname	Object name

[Return Parameters]

ID	～id	Specified object ID
	or	Error Code

[Error Codes]

E_ID	Invalid ID number (domid is invalid or cannot be used)
E_NOEXS	Object does not exist (object of oname does not exist)
E_PAR	Parameter error (oname is invalid or cannot be used)

[Description]

Retrieves the object that belongs to the domain shown by domid by using the object name, and acquires the object ID.

The object name of the object to be retrieved is specified in oname.

When the object specified by domid and oname is found, the ID of the object is returned. When the corresponding object does not exist, E_NOEXS is returned.

Objects that can be retrieved are accessible objects only. For inaccessible objects due to access protection, retrieval fails and E_NOEXS is returned.

[Items Concerning SMP T-Kernel]

This call does not exist in the T-Kernel 1.00 Specification.

tk_dmi_dom, tk_dmi_tsk, tk_dmi_sem,tk_dmi_flg, tk_dmi_mbx, tk_dmi_mtx, tk_dmi_mbf, tk_dmi_por, tk_dmi_mpf, tk_dmi_mpl, tk_dmi_alm, tk_dmi_cyc

Get Domain Information of Each Object

tk_dmi_xxx: Get Domain Information

[C Language Interface]

```
ER ercd = tk_dmi_dom ( ID domid, TD_DMI *pk_dmi ) ;     /* Domain */
ER ercd = tk_dmi_tsk ( ID tskid, TD_DMI *pk_dmi ) ;     /* Task */
ER ercd = tk_dmi_sem ( ID semid, TD_DMI *pk_dmi ) ;     /* Semaphore */
ER ercd = tk_dmi_flg ( ID flgid, TD_DMI *pk_dmi ) ;     /* Event Flag */
ER ercd = tk_dmi_mbx ( ID mbxid, TD_DMI *pk_dmi ) ;     /* Mailbox */
ER ercd = tk_dmi_mtx ( ID mtxid, TD_DMI *pk_dmi ) ;     /* Mutex */
ER ercd = tk_dmi_mbf ( ID mbfid, TD_DMI *pk_dmi ) ;     /* Message Buffer */
ER ercd = tk_dmi_por ( ID porid, TD_DMI *pk_dmi ) ;     /* Rendezvous Port */
ER ercd = tk_dmi_mpf ( ID mpfid, TD_DMI *pk_dmi ) ;     /* Fixed-size Memory Pool */
ER ercd = tk_dmi_mpl ( ID mplid, TD_DMI *pk_dmi ) ;     /* Variable Length Memory Pool */
ER ercd = tk_dmi_alm ( ID almid, TD_DMI *pk_dmi ) ;     /* Alarm Handler */
ER ercd = tk_dmi_cyc ( ID cycid, TD_DMI *pk_dmi ) ;     /* Cyclic Handler */
```

[Parameters]

ID	～id	ObjectID	Object ID
TD_DMI*	pk_dmi	Packet to Domain Information	Packet address where domain information is returned

[Return Parameters]

ER ercd	ErrorCode	Error code

pk_dmi details

ATR	domatr	DomainAttribute	Domain attribute
ID	domid	DomainID	ID of the domain to which it belongs
ID	kdmid	Kernel Domain ID	ID Kernel domain ID to which it belongs
UB	oname[8]	Object name	Object name

[Error Codes]

E_ID	Invalid ID number (~id is invalid or cannot be used)
E_NOEXS	Object does not exist (object of ~id does not exist)
E_PAR	Parameter error (alue for which the packet address for the return parameter cannot be used)
E_DACV	Access protection violation

[Description]

Gets information related to the domain of the target object displayed by ~id.

The attribute related to the domain of the target object is set in domatr. domatr takes the following values.

domatr := [TA_ONAME]|(TA_PRIVATE || TA_PROTECTED || TA_PUBLIC)

TA_ONAME	Object name is specified
TA_PROTECTED	Access protection attribute is protect
TA_PRIVATE	Access protection attribute is private
TA_PUBLIC	Access protection attribute is public

The ID number of the domain to which the target object belongs is set in domid.
The ID number of the kernel domain to which the target object belongs is set in kdmid.
The object name of the target object is set to oname. When the object name is not set to the target object, all contents are 0.
Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

This call does not exist in the T-Kernel 1.00 Specification.

Get Kernel Domain ID tk_get_kdm

tk_get_kdm: Get Kernel DomainID

[C Language Interface]

```
ID kdmid = tk_get_kdm ( ID prcid ) ;
```

[Parameters]

ID	prcid	ProcessorID	Processor ID

[Return Parameters]

ID	kdmid	Kernel DomeinID	Kernel Domain ID

[Error Codes]

E_ID	Invalid ID number (prcid is invalid or cannot be used)

[Description]
Gets the ID number of the kernel domain of the SMP T-Kernel that is operated by the processor shown by prcid.
When prcid = PRC_SELF = 0, its own kernel domain is returned.

[Additional Notes]
Since there is only one kernel domain in SMP T-Kernel, the ID number of the same domain is always acquired in the same system.

[Items Concerning SMP T-Kernel]
This call does not exist in the T-Kernel 1.00 Specification.

Reference Domain Status tk_ref_dom

tk_ref_dom: Refer Domain Status

[C Language Interface]

ER ercd = tk_ref_dom (ID domid, T_RDOM *pk_rdom) ;

[Parameters]

ID	domid	DomainID	Domain ID
T_RDOM*	pk_rdom	Packet to Refer Domain	Packet address to where the domain status is returned

[Return Parameters]

ER	ercd	ErrorCode	Error Code

pk_rdom details

VP	exinf	ExtendedInformation	Extended information
ID	domid	DomainID	Domain ID to which the domain belongs
UINT	objcnt	ObjectCount	Object count that belongs to the domain

——(Other implementation-dependent parameters may be added beyond this point)——

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (domid is invalid or cannot be used)
E_NOEXS	Object does not exist (domain of domid does not exist)
E_PAR	Parameter error (Value for which the packet address for the return parameter cannot be used)

[Description]

References the various target domain statuses shown by domid, and returns extended information (exinf), the domain ID (domid) to which the target domain belongs, and the object count which belongs to the target domain as return parameters.

Only objects that belong directly to the target domain are included in the object count (objcnt). Although the domain (subdomain) that belongs to the target domain is included in the object count, the object which belongs to the subdomain is not included in the number. That is, the object (including when it belongs without being specified by default) that specifies the target domain as the domain to which the object belongs during creation becomes the target.

[Items Concerning SMP T-Kernel]

This call does not exist in the T-Kernel 1.00 Specification.

4.9 Interrupt Management Functions

Interrupt management functions are for defining and manipulating handlers for external interrupts and CPU exceptions.

An interrupt handler runs as a task-independent portion. System calls can be invoked in a task-independent portion in the same way as in a task portion, but the following restriction applies to system call issuing in a task-independent portion.

- A system call that implicitly specifies the invoking task, or one that may put the invoking task in WAIT state cannot be issued. Error code E_CTX is returned in such cases.

In SMP T-Kernel, interrupt handlers are executed on the processor where external interrupts or CPU exceptions occur.

During task-independent portion execution, task switching (dispatching) does not occur. If system call processing results in a dispatch request, the dispatch is delayed until processing leaves the task-independent portion. This is called delayed dispatching. However, dispatch is possible in other processors than processors executing the interrupt handler. Therefore, what is delayed is the dispatch by the processor executing the interrupt handler.

Each system call is summarized in the table below. For more details refer to the explanation for each system call.

Call Name	Function	Different from T-Kernel 1.00 Specification
tk_def_int	Define Interrupt Handler	○
tk_ret_int	Return from interrupt handler	○

Different from T-Kernel 1.00 Specification ○:No ×:Yes △: Different only in that E_DACV is returned due to the access protection

tk_def_int: Define Interrupt Handler

[C Language Interface]

ER ercd = tk_def_int (UINT dintno, T_DINT *pk_dint) ;

[Parameters]

UINT	dintno	InterruptDefineNumber	Interrupt definition number
T_DINT*	pk_dint	Packet to Define InterruptHandler	Packet of interrupt handler definition information

pk_dint detail:

ATR	intatr	InterruptHandlerAttribute	Interrupt handler attributes
FP	inthdr	InterruptHandlerAddress	Interrupt handler address
UW	assprc	AssignProcessor	Execution processor specification

──(Other implementation-dependent parameters may be added beyond this point.)──

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_NOMEM	Insufficient memory (memory for control block cannot be allocated)
E_RSATR	Reserved attribute (intatr is invalid or cannot be used)
E_PAR	Parameter error (dintno, pk_dint, or inthdr is invalid or cannot be used)

[Description]

Here “interrupts” include both external interrupts from a device and CPU exceptions.

Defines an interrupt handler for interrupt definition number dintno, and enables the use of the interrupt handler. This system call maps the interrupt definition number indicated in dintno to the interrupt handler address and attributes.

The specific signicance of dintno is defined separately for each implementation, but generally it means an interrupt vector number.

intatr indicates system attributes in its low bits, with the high bits used for implementation-dependent attributes. The system attributes part of intatr is specified in the following format.

intatr := (TA_ASM ││ TA_HLNG) | [TA_ASSPRC]

TA_ASM	The handler is written in assembly language
TA_HLNG	The handler is written in a high-level language
TA_ASSPRC	Specifies the execution processor

```
#define TA_ASM       0x00000000   /* assembly program */
#define TA_HLNG      0x00000001   /* high-level language program */
#define TA_ASSPRC    0x00020000   /* execution processor specification */
```

When the TA_ASM attribute is specified, in principle the OS is not involved in interrupt handler starting. When an interrupt is raised, the interrupt handling system of the CPU hardware (depending on the implementation, processing by T-Monitor may be included) directly starts the interrupt handler defined by this system call. Accordingly, processing for saving and restoring registers used by the interrupt handler is necessary at the beginning and end of the interrupt handler. An interrupt handler is terminated by execution of the tk_ret_int system call or by the CPU interrupt return instruction (or equivalent means).

Provision of a means for return from an interrupt handler without using tk_ret_int and without OS intervention is mandatory. Note that if tk_ret_int is not used, delayed dispatching is not necessary.

Support for return from an interrupt handler using tk_ret_int is mandatory, and in this case, delayed dispatching is necessary.

When the TA_HLNG attribute is specified, the interrupt handler is started via a high-level language support routine. The high-level language support routine takes care of saving and restoring register values. The interrupt handler terminates by a simple return from the function. The interrupt handler takes the following format when the TA_HLNG attribute is specified.

```
void inthdr( UINT dintno )
{
        /*
                Processing
        */
        return; /* exit interrupt handler */
}
```

The parameter dintno passed to an interrupt handler is a number identifying the interrupt that was raised, and is the same as that specified with tk_def_int. Depending on the implementation, other information about the interrupt may be passed in addition to dintno. If such information is used, it must be defined for each implementation in a second parameter or subsequent parameters passed to the interrupt handler.

If the TA_HLNG attribute is specified, it is assumed that the CPU interrupt flag will be set to interrupts disabled state from the time the interrupt is raised until the interrupt handler is called. In other words, as soon as an interrupt is raised, the state goes to multiple interrupts disabled, and this state remains when the interrupt handler is called. If multiple interrupts are enabled, the interrupt handler must include processing that enables interrupts by manipulating the CPU interrupt flag.

Also in the case of the TA_HLNG attribute, upon entry into the interrupt handler, system call issuing must be possible. Note, however, that assuming standard provision of the functionality described above, extensions such as adding a function for entering an interrupt handler with multiple interrupts enabled are allowed.

When the TA_ASM attribute is specified, the state upon entry into the interrupt handler is separately defined for each implementation. Issues such as the status of the stack and registers upon interrupt handler entry, whether system calls can be made, the method of invoking system calls, and the method of returning from the interrupt handler without OS intervention must all be defined explicitly.

In the case of the TA_ASM attribute, depending on the implementation, there may be cases where interrupt handler execution is not considered to be a task-independent portion. In such a case, the following points need to be noted carefully.

- If interrupts are enabled, there is a possibility that task dispatching will occur.
- When a system call is invoked, it will be processed as having been called from a task portion or quasi-task portion.

If a method is provided for performing some kind of operation in an interrupt handler to have it detected as task-independent portion, that method must be indicated for each implementation.

Whether the TA_HLNG or TA_ASM attribute is specified, upon entry into an interrupt handler, the logical space at the time the interrupt occurred is retained. No processing takes place upon return from the interrupt handler for restoring the logical space to its state at the time the interrupt was raised. Switching logical spaces inside the interrupt handler is not prohibited, but the OS is not aware of the effects of logical space switching.

In case of the TA_ASSPRC attribute, assprc becomes valid and the processor that executes the interrupt handler is set. However, whether or not the TA_ASSPRC attribute is specifiable is implementation-defined. Moreover, even if the TA_ASSPRC attribute is specifiable, the processor where the interrupt handler can be set is also implementation-defined. When invalid specification is conducted, an E_PAR error is returned. The values of assprc are defined as follows.

```
#define TP_PRC1     0x00000001     Processor ID Number 1 Processor
#define TP_PRC2     0x00000002     Processor ID Number 2 Processor
#define TP_PRC3     0x00000004     Processor ID Number 3 Processor
#define TP_PRC4     0x00000008     Processor ID Number 4 Processor
                        ~
#define TP_PRC32    0x80000000     Processor ID Number 32 Processor
```

Whether or not multiple processors can be specified to execute the interrupt handler is implementation-defined. When multiple specifications are possible, the value of the logical sum of the target processor is specified for assprc. For example, to specify ID number 1 and ID number 2 processors, the value of the logical sum of TP_PRC1 and TP_PRC2 is specified.

When TA_ASSPRC is not specified, assprc is ignored. In this case, the processor that executes the interrupt handler is implementation-defined.

During execution of the interrupt handler, even if a dispatch becomes necessary, priority is first given to execute the interrupt handler until completion, and dispatch is delayed until the completion of the interrupt handler (principle of delay dispatching). However, dispatch is possible in other processors than the processor executing the interrupt handler. Therefore, what is delayed is the dispatch by the processor executing the interrupt handler.

An interrupt handler runs as a task-independent portion. As such, it is not possible in an interrupt handler, to call a system call that can enter WAIT state, or one that is intended for the invoking task.

When pk_dint = NULL is set, a previously defined interrupt handler is canceled. When the handler for an interrupt is canceled, the default handler defined by T-Monitor is used.

It is possible to redefine an interrupt handler for an interrupt number already having a defined handler. It is not necessary first to cancel the definition for that number. Defining a new handler for a dintno already having an interrupt handler defined does not return an error.

[Additional Notes]

The various specifications governing the TA_ASM attribute are mainly concerned with achieving an interrupt hook. For example, when an exception is raised due to an illegal address access, ordinarily an interrupt handler defined in a higher-level program detects this and performs the error processing; but in the case of debugging, in place of error processing by a higher-level program, a T-Monitor interrupt handler does the processing and starts a debugger. In this case, the interrupt handler defined by the higher-level program hooks the T-Monitor interrupt handler. After that, depending on the situation, either interrupt handling is passed off to T-Monitor or the other program does the processing on its own.

The interrupt handler is executed on the processor where the interrupt occurs due to hardware function. Therefore, setting the execution processor for the interrupt handler is to set the processor where the interrupt occurs. In other words, the setting function of the interrupt handler's execution processor by the TA_ASSPRC attribute is for standardizing the method for setting the processor to process the interrupt in SMP. This is largely depends on hardware. It is impossible to cause interrupts with a random processor and it is possible that interrupts always occur in a specific processor as well. Therefore, the setting of the execution processor including the availability is implementation-defined.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

- TA_ASSPRC is added to the interrupt handler attribute and the processor for executing the interrupt handler is specifiable.

tk_ret_int: Return from Interrupt Handler

[C Language Interface]

void tk_ret_int (void) ;

Although this system call is defined in the form of a C language interface, it will not be called in this format if a high-level language support routine is used.

[Parameters]

None.

[Return Parameters]

 Does not return to the context issuing the system call.

[Error Codes]

 The following kind of error may be detected, but no return is made to the context issuing the system call even if the error is detected. For this reason, the error code cannot be passed directly as a system call return parameter. If an error is detected, the behavior is implementation-dependent.

E_CTX Context error (issued from other than an interrupt handler (implementation-dependent error)

[Description]

Exits an interrupt handler.

During execution of the interrupt handler, dispatch does not occur in the processor, and is delayed until the interrupt handler exits by tk_ret_int (principle of delayed dispatching). Therefore, the delayed dispatch requests are collectively processed in tk_ret_int.

tk_ret_int is invoked only if the interrupt handler was defined specifying the TA_ASM attribute. In the case of a TA_HLNG attribute interrupt handler, the functionality equivalent to tk_ret_int is executed implicitly in the high-level language support routine, so tk_ret_int is not (must not be) called explicitly.

As a rule, the OS is not involved in the starting of a TA_ASM attribute interrupt handler. When an interrupt is raised, the defined interrupt handler is started directly by the interrupt system of the CPU hardware. The saving and restoring of registers used by the interrupt handler must therefore be handled in the interrupt handler.

For the same reason, the stack and register states at the time tk_ret_int is issued must be the same as those at the time of entry into the interrupt handler. Because of this, in some cases function codes cannot be used in tk_ret_int, in this case, tk_ret_int may be implemented using a trap instruction with a vector different from that used for other system calls.

[Additional Notes]

tk_ret_int is a system call that does not return to the context from which it was called. Even if an error code is returned when an error of some kind is detected, normally no error checking is performed in the context from which the system call was invoked, leaving the possibility that the program will hang. For this reason, this system call does not return even if error is detected.

Using an assembly-language return (REIT) instruction instead of tk_ret_int to exit the interrupt handler is possible if it is clear no dispatching will take place on return from the handler (the same task is guaranteed to continue executing), or if there is no need for dispatching to take place.

Depending on the CPU architecture and method of configuring the OS, it may be possible to perform delayed dispatching even when an interrupt handler exits using an assembly-language REIT instruction. In such cases, the assembly-language REIT instruction may be interpreted as though it were a tk_ret_int system call.

Performing of E_CTX error checking when tk_ret_int is called from a time event handler is implementation-dependent. Depending on the implementation, control may return from a different type of handler.

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

4.10 System Management Functions

System management functions are functions for changing and referencing system states. Functions are provided for rotating task precedence in a queue, getting the ID of the task in RUN state, disabling and enabling task dispatching, referencing context and system states, setting low-power mode, and referencing the T-Kernel version.

In SMP T-Kernel, each system call for system state management functions are functionally the same as the T-Kernel 1.00 Specification. However, operations may differ because multiple tasks in RUN state exist and each can go to a different state.
Each system call is summarized in the table below. For more details refer to the explanation for each system call.

Call Name	Function	Different from T-Kernel 1.00 Specification
tk_rot_rdq	Rotate task queue	○
tk_get_tid	Get Task Identifier	×
tk_dis_dsp	Disable Dispatch	×
tk_ena_dsp	Enable Dispatch	×
tk_get_prc	Get Executing Processor ID	※
tk_ref_sys	Reference System Status	×
tk_set_pow	Set Power Mode	○
tk_ref_ver	Reference Version Information	×

Different from T-Kernel 1.00 Specification ○:No ×:Yes △: Different only in that E_DACV is returned due to the access protection

※Since it is a new system call, it does not exist in the T-Kernel 1.00 specification.

tk_rot_rdq: Rotate Ready Queue

[C Language Interface]

```
ER ercd = tk_rot_rdq ( PRI tskpri ) ;
```

[Parameters]

PRI	tskpri	TaskPriority	Task priority

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_PAR	Parameter error (tskpri is invalid)

[Description]

Rotates the precedence among tasks having the priority specified in tskpri. This system call changes the precedence of tasks in RUN or READY state having the specified priority, so that the task with the highest precedence among those tasks is given the lowest precedence.

When TPRI_RUN(=0) is specified in tskpri, the RUN state (RUNNING) task priority at that time is the target priority and the task precedence of the priority rotates. When tk_rot_rdq(tskpri=TPRI_RUN) is issued from the task, the priority of the invoking task is the target priority.

tk_rot_rdq(tskpri=TPRI_RUN) can also be issued from the task-independent portion of the cyclic handler, etc. In this case, the priority of the task with the highest precedence is the target priority.

[Additional Notes]

If there are no tasks in a run state having the specified priority, or only one such task, the system call completes normally with no operation (no error code is returned).

When this system call is issued in dispatch enabled state, specifying as the priority either TPRI_RUN or the current priority of the invoking task as the priority, the precedence of the invoking task will become the lowest among tasks of the same priority. In this way, the system call can be used to relinquish execution privilege.

In dispatch disabled state, the task with highest precedence among tasks of the same priority is not always the currently executing task. The precedence of the invoking task will therefore not always become the lowest among tasks having the same priority when the above method is used in dispatch disabled state.

Examples of tk_rot_rdq execution are given in Figure 20(a) and Figure 20(b). When this system call is issued in the state shown in Figure 20(a) specifying tskpri = 2, the new precedence order becomes that in Figure 20(b) and Task C becomes the executing task.

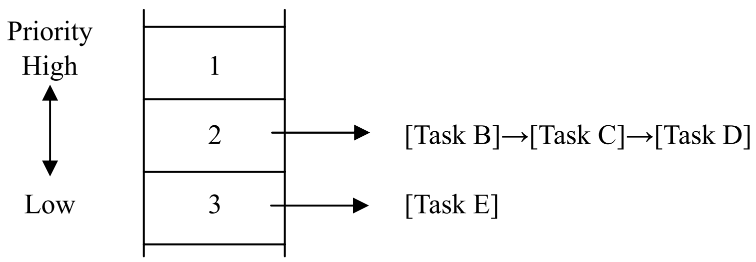


Figure 20(a): Precedence Before Issuing tk_rot_rdq

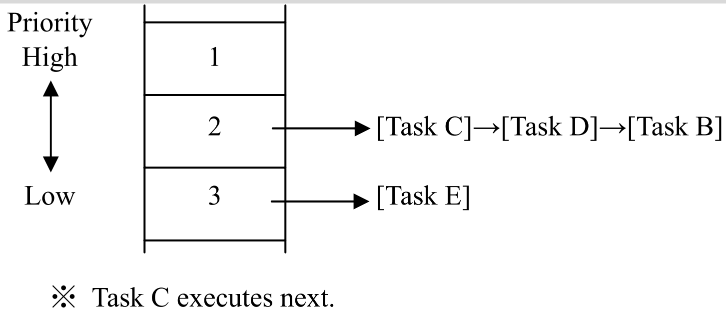


Figure 20(b): Precedence After Issuing tk_rot_rdq (tskpri = 2)

[Items Concerning SMP T-Kernel]

There are no differences in the functions of this call with the T-Kernel 1.00 Specification. However, in SMP T-Kernel, even if tk_rot_rdq() is executed to make the priority of the RUN state task lower than other tasks, the RUN state does not always end because multiple tasks can be executed at the same time. This is the same as when the task execution continues even if tk_rot_rdq() is executing when the task that can be executed in the T-Kernel 1.00 Specification is one.

Moreover, when TPRI_RUN(=0) is specified in tskpri, although one task will definitely become an object in the T-Kernel 1.00 Specification, one task among multiple tasks is selected because multiple tasks operate at the same time in SMP T-Kernel. When this call is issued from the task, the execution order of the invoking task becomes the lowest among tasks with the same priority, and execution privilege is abandoned. When this call is issued from the task-independent portion, the task with the highest precedence becomes the lowest among tasks with the same priority.

tk_get_tid:Get Task Identifier

[C Language Interface]

```
ID tskid = tk_get_tid ( void ) ;
```

[Parameters]

None

[Return Parameters]

ID	tskid	TaskID	ID of the task in RUN state

[Error Codes]

None

[Description]

Gets the ID number of the task currently in RUN state.

When this call is issued from the task, the task ID of the invoking task can be acquired. Unless the task-independent portion is executing, the current RUN state task will be the invoking task.

If there is no task currently in RUN state, 0 is returned.

[Additional Notes]

The task ID returned by tk_get_tid is identical to runtskid returned by tk_ref_sys.

[Items Concerning SMP T-Kernel]

When this call is issued from the task, the ID number of the invoking task is acquired as in the T-Kernel 1.00 Specification. When it is issued from the task-independent portion, the ID number of the task that was executing immediately prior to the execution of the task-independent portion is acquired in the processor that issued this call. When there are no tasks in RUN state prior to the execution of the task-independent portion in the processor, 0 is returned, however, there is a possibility of a RUN state existing in another processor.

tk_dis_dsp: Disable Dispatch

[C Language Interface]

```
ER ercd = tk_dis_dsp ( void ) ;
```

[Parameters]

None

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_CTX	Context error (issued from task-independent portion)

[Description]

Disables task dispatching. Dispatch disabled state remains in effect until tk_ena_dsp is called to enable task dispatching. While dispatching is disabled, the invoking task does not change from RUN state to READY state or to WAIT state. External interrupts, however, are still enabled, so even in dispatch disabled state an interrupt handler can be started. In dispatch disabled state, the running task can be preempted by an interrupt handler, but not by another task.

The specific operations during dispatch disabled state are as follows.

- Even if a system call issued from an interrupt handler or by the task that called tk_dis_dsp results in a task going to READY state with a higher priority than the task that called tk_dis_dsp, that task will not be dispatched.
  In SMP T-Kernel, if a task in dispatch disabled state exists during task scheduling, the processor that executes the task is excluded from the target of scheduling. Therefore, tasks in dispatch disabled state are always executed by the same processor and other tasks are not executed by this processor during this time. Even when a task is preempted by an interruption handler, etc., the task continues to be executed with the same processor after the handler ends.
- If the task that called tk_dis_dsp issues a system call that may cause the invoking task to be put in WAIT state (e.g., tk_slp_tsk or tk_wai_sem), error code E_CTX is returned to it.
- When system status is referenced by tk_ref_sys, TSS_DDSP is returned in sysstat.

If tk_dis_dsp is called for a task already in dispatch disabled state, that state continues with no error code returned. No matter how many times tk_dis_dsp is called, calling tk_ena_dsp just once is sufficient to enable dispatching again. The operation when the pair of system calls tk_dis_dsp and tk_ena_dsp are nested must therefore be managed by the user as required.

[Additional Notes]

A task in RUN state cannot go to DORMANT state or NON-EXISTENT state while dispatching is disabled. If tk_ext_tsk or tk_exd_tsk is called for a task in RUN state while interrupts or dispatching is disabled, error code E_CTX is detected. Since, however, tk_ext_tsk and tk_exd_tsk are system calls that do not return to their original context, such errors are not passed in return parameters by these system calls.

[Items Concerning SMP T-Kernel]

Dispatch is prohibited by this call only for the task issuing the call, and preemption is not conducted from other tasks. In SMP T-Kernel, multiple tasks are executed at the same time. Dispatch is permitted for tasks other than the task that issued this call. Therefore, task dispatch occurs in SMP T-Kernel. In addition, the dispatch disabled state can be thought of as a specific task status and not the status of SMP T-Kernel overall.

It is guaranteed that tasks in dispatch disabled state will be executed on the same processor. If there is a need to continue in RUN state only without fixing an execution processor, the task priority (increasing the priority higher than other tasks) must be changed instead of dispatch prohibition.

In the T-Kernel 1.00 Specification, by use of that tasks which are executed at the same time do not exist, and dispatch prohibition was used in exclusive control between tasks. However, in SMP T-Kernel, this is impossible because other tasks are executed even if they are dispatch disabled state or because dispatch also occurs.

tk_ena_dsp: Enable Dispatch

[C Language Interface]

```
ER ercd = tk_ena_dsp ( void ) ;
```

[Parameters]

None

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_CTX	Context error (issued from task-independent portion)

[Description]

Enables task dispatching. This system call cancels the disabling of dispatching by the tk_dis_dsp system call.

If tk_ena_dsp is called for a task not in dispatch disabled state, the dispatch enabled state continues and no error code is returned.

[Items Concerning SMP T-Kernel]

Dispatch is permitted only for tasks that execute this call. In SMP T-Kernel, multiple tasks are executed at the same time and each respective task can become dispatch disabled. When this call is executed, there is no affect even if other tasks are dispatch disabled.

Get Executing Processor ID tk_get_prc

tk_get_prc: Get Processor Identifier

[C Language Interface]

```
ID prcid = tk_get_prc ( void ) ;
```

[Parameters]

None

[Return Parameters]

ID	prcid	Processor ID	Executing Processor ID

[Error Codes]

None

[Description]

Gets the Processor ID of the processor executing the program that issued this call.

[Additional Notes]

In SMP T-Kernel, when this call is issued from the task portion, the acquired processor ID is the one when this call was issued. Since there is a possibility a dispatch has occurred prior to the return from this call, it cannot be guaranteed that the acquired processor ID is for the currently executing processor except when the task is executed by a specific processor due to reasons such as execution processor specification and dispatch prohibition.

[Items Concerning SMP T-Kernel]

This call is a new system call that is not in the T-Kernel 1.00 Specification.

tk_ref_sys: Refer System Status

[C Language Interface]

```
ER ercd = tk_ref_sys ( T_RSYS *pk_rsys ) ;
```

[Parameters]

T_RSYS* pk_rsys	Packet to Refer System	Address of packet for returning status information

[Return Parameters]

ER	ercd	ErrorCode	Error code

pk_rsys detail:

INT	sysstat	SystemState	System status
ID	runtskid		ID of task currently in RUN state
ID	schedtskid		ID of task scheduled to run next

――(Other implementation-dependent parameters may be added beyond this point.)――

[Error Codes]

E_OK	Normal completion
E_PAR	Parameter error (the return parameter packet address cannot be used)

[Description]

Gets the current system execution status, passing in return parameters system information such as the dispatch disabled state and whether a task-independent portion is executing.

In SMP T-Kernel, information on the processor which issued this call is acquired.

The following values are returned in sysstat.

```
sysstat := ( TSS_TSK  | [TSS_DDSP] | [TSS_DINT] )
         || ( TSS_QTSK | [TSS_DDSP] | [TSS_DINT] )
         || ( TSS_INDP )
```

TSS_TSK	0	Task portion executing
TSS_DDSP	1	Dispatch disabled
TSS_DINT	2	Interrupts disabled
TSS_INDP	4	Task-independent portion executing
TSS_QTSK	8	Quasi-task portion executing

The ID of the task currently in RUN state is returned in runtskid, while schedtskid indicates the ID of the next task scheduled to go to RUN state. Normally runtskid = schedtskid, but this is not necessarily true if, for example, a higher-priority task was wakened during dispatch disabled state. If there is no such task, 0 is returned.

It must be possible to invoke this system call from an interrupt handler or time event handler.

[Additional Notes]

Depending on the OS implementation, the information returned by tk_ref_sys is not necessarily guaranteed to be accurate at all times.

[Items Concerning SMP T-Kernel]

In SMP T-Kernel, there exists a RUN state for each respective processor. Therefore, information on the processor that issued this call is returned. Information on other processors cannot be acquired.

tk_set_pow: Set Power Mode

[C Language Interface]

```
ER ercd = tk_set_pow ( UINT powmode ) ;
```

[Parameters]

UINT	powmode	Low-power mode

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_PAR	Parameter error (value that cannot be used in powmode)
E_QOVR	Low-power mode disable count overflow
E_OBJ	TPW_ENALOWPOW was requested with low-power mode disable count at 0

[Description]
The following two power-saving functions are supported.

● Switching to low-power mode when the system is idle
When there are no tasks to be executed, the system switches to a low-power mode provided in hardware.
Low-power mode is a function for reducing power use during very short intervals, such as from one timer interrupt to the next. This is accomplished, for example, by lowering the CPU clock frequency. It does not require complicated mode-switching in software but is implemented mainly using hardware functionality.

● Automatic power-off
When the operator performs no operations for a certain length of time, the system automatically cuts the power and goes to suspended state. If there is a start request (interrupt, etc.) from a peripheral device or if the operator turns on the power, the system resumes from the state when the power was cut.
In the case of a power supply problem such as low battery, the system likewise cuts the power and goes to suspended state.
In suspended state, the power is cut to peripheral devices and circuits as well as to the CPU, but the main memory contents are retained.

tk_set_pow sets the low-power mode. The power-saving functions (off_pow, low_pow) of T-Kernel/SM are called in the power-saving mode according to the setting.

powmode:= (TPW_DOSUSPEND || TPW_DISLOWPOW || TPW_ENALOWPOW)

#define TPW_DOSUSPEND	1	Suspended state
#define TPW_DISLOWPOW	2	Switching to low-power mode disabled
#define TPW_ENALOWPOW	3	Switching to low-power mode enabled (default)

・TPW_DOSUSPEND
Execution of all tasks and handlers is stopped, peripheral circuits (timers, interrupt controllers, etc.) are stopped, and the power is cut (suspended). (off_pow is called.)
When power is turned back on, peripheral circuits are restarted, execution of all tasks and handlers is resumed, operations resume from the point before power was cut, and the system call returns.
If for some reason the resume processing fails, normal startup processing (for reset) is performed and the system boots fresh.

・TPW_DISLOWPOW
Switching to low-power mode in the dispatcher is disabled. (low_pow is not called.)

・TPW_ENALOWPOW

Switching to low-power mode in the dispatcher is enabled. (low_pow is called).

The default at system startup is low-power mode enabled (TPW_ENALOWPOW).

Each time TPW_DISLOWPOW is specified, the request count is taken. Low-power mode is enabled only when TPW_ENALOWPOW is requested for as many times as TPW_DISLOWPOW was requested. The maximum request count is implementation-dependent, but a count of at least 255 times must be possible.

[Additional Notes]

off_pow and low_pow are T-Kernel/SM functions. See 5.6 for details.

T-Kernel does not detect power supply problems or other factors for suspending the system. Actual suspension requires suspend processing in each of the peripheral devices (device drivers). The system is suspended not by calling tk_set_pow directly but by use of the T-Kernel/SM suspend function.

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

Reference Version Information tk_ref_ver

tk_ref_ver: Refer Version Information

[C Language Interface]

```
ER ercd = tk_ref_ver ( T_RVER *pk_rver ) ;
```

[Parameters]

T_RVER*	pk_rver	Packet of Version Information	Start address of packet for version information

[Return Parameters]

ER	ercd	Error Code	Error code

pk_rver detail:

UH	maker	Maker	T-Kernel maker code
UH	prid	Product ID	T-Kernel ID
UH	spver	Specification Version	Specification version
UH	prver	Product Version	T-Kernel version
UH	prno[4]	Product Number	T-Kernel products management information

[Error Codes]

E_OK	Normal completion
E_PAR	Parameter error (the address of the return parameter packet cannot be used)

[Description]

Gets information about the T-Kernel version in use, returning that information in the packet specified in pk_rver. The following information can be obtained.

maker is the vendor code of the T-Kernel implementing vendor. The maker field has the format shown in Figure 21(a).

[maker]

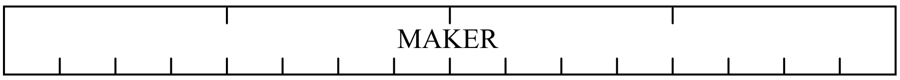


Figure 21(a): maker Field Format

prid is a number indicating the T-Kernel type. The prid format is shown in Figure 21(b).

Assignment of values to prid is left up to the vendor implementing T-Kernel. Note, however, that this is the only number distinguishing product types, and that vendors should give careful thought to how they assign these numbers, doing so in a systematic way. In that way the combination of maker code and prid becomes a unique identifier of the T-Kernel type.

The reference code of SMP T-Kernel is provided by the T-Engine Forum, and the maker and prid are as follows.

maker = 0x0000
prid = 0x0000

[prid]

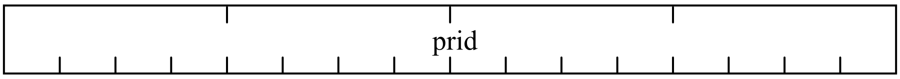


Figure 21(b): prid Field Format

The upper 4 bits of spver give the TRON specification series. The low 12 bits indicate the T-Kernel specification version implemented. The format of spver is shown in Figure 21(c).

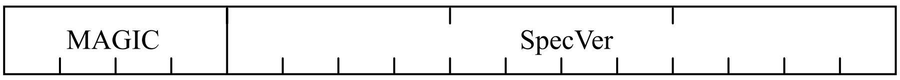
If, for example, a product conforms to the T-Kernel specification Ver 1.02.xx, spver is as follows.

MAGIC	= 0x5	(SMP T-Kernel)
SpecVer	= 0x102	(Ver 1.02)
spver	= 0x5102	

If a product implements the SMP T-Kernel specification draft version Ver 1.B0.xx, spver is as follows.

MAGIC	= 0x5	(SMP T-Kernel)
SpecVer	= 0x1B0	(Ver 1.B0)
spver	= 0x51B0	

[spver]



MAGIC: Types of OS Specifications
- 0x0 TRON common (TAD, etc.)
- 0x1 reserved
- 0x2 reserved
- 0x3 reserved
- 0x4 AMP T-Kernel
- 0x5 SMP T-Kernel
- 0x6 μT-Kernel
- 0x7 T-Kernel

SpecVer: The version of the specification on which the kernel complies with. This is given as a three-digit packed-format BCD code. In the case of a draft version, the letter A, B, or C may appear in the second digit. In this case, the corresponding hexadecimal form of A, B, or C is inserted.

Figure 21(c): spver Field Format

prver is the version number of the T-Kernel implementation. The specific values assigned to prver are left to the vendor implementing the T-Kernel to decide.

prno is a return parameter for use in indicating T-Kernel product management information, product number, etc. The specific meaning of values set in prno is left to the vendor implementing T-Kernel to decide.

[Additional Notes]

The format of the packet and structure members for getting version information is mostly uniform across the various TRON specifications, but the CPU information and variation descriptors are not specified.

The value obtained by tk_ref_ver in SpecVer is the first three digits of the specification version number. The numbers after that indicate minor revisions such as those issued to correct misprints and the like, and are not obtained by tk_ref_ver. For the purpose of matching to the specification contents, the first three numbers of the specification version are sufficient.

An OS implementing a draft version may have A, B, or C as the second number of SpecVer. It must be noted that in such cases the specification order of release may not correspond exactly to higher and lower SpecVer values. For example, specifications may be released in the following order:

Ver 1.A1 → Ver 1.A2 → Ver 1.B1 → Ver 1.C1 → Ver 1.00 → Ver 1.01 → …

In this example, when going from Ver 1.Cx to Ver 1.00, SpecVer goes from a higher to a lower value.

[Items Concerning SMP T-Kernel]

Differences from the T-Kernel 1.00 Specification are as follows.

・AMP T-Kernel and SMP T-Kernel were added to MAGIC which shows the type of OS specification.

4.11 Subsystem Management Functions

Subsystems are programs that realize extended system calls (extended SVC), and conduct the functional extensions of T-Kernel. Subsystems are under the management of T-Kernel and the interface between T-Kernel and subsystems is stipulated by the T-Kernel Specification.

Users can implement the subsystem, define the extended SVCs, and uniquely extend the functions of T-Kernel. Moreover, T-Kernel/SM and Standard Extension, etc. have been realized by the use of subsystems in system software as well.

In SMP T-Kernel, there are no differences in the subsystem management function with the T-Kernel 1.00 Specification. Each system call is summarized in the table below. For more details refer to the explanation for each system call.

Call Name	Function	Different from T-Kernel 1.00 Specification
tk_def_ssy	Define Subsystem	○
tk_sta_ssy	Call Startup Function of Subsystem	○
tk_cln_ssy	Call Cleanup Function of Subsystem	○
tk_evt_ssy	Call Event Function of Subsystem	○
tk_ref_ssy	Reference Subsystem Status	○
tk_cre_res	Create Resource Group	○
tk_del_res	Delete Resource Group	○
tk_get_res	Get Resource Management Block	○

Different from T-Kernel 1.00 Specification ○:No ×:Yes △: Different only in that E_DACV is returned due to the access protection

Subsystems consist of the extended SVC handler to accept the extended SVC, each function (break function/start-up function/cleanup function/event function) for accepting requests by T-Kernel, and resource control blocks (Figure 22).

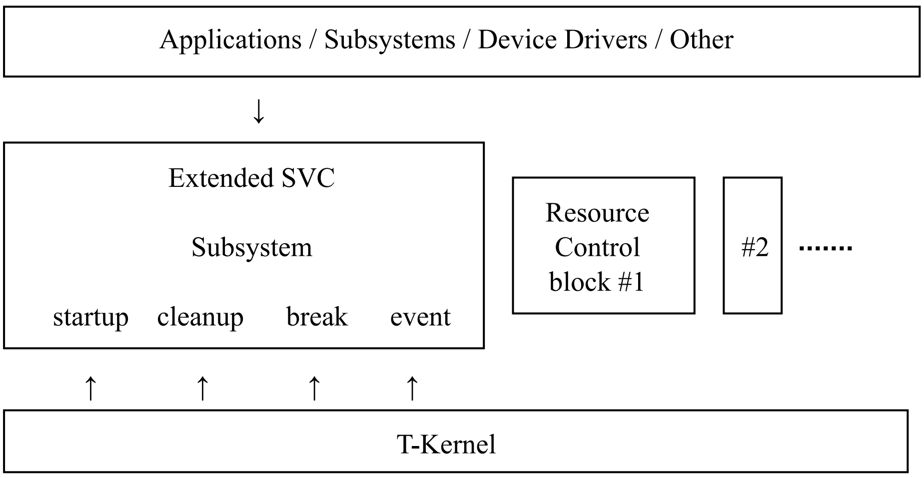


Figure 22: T-Kernel Subsystems

The resource control block is a memory area for managing resources that the subsystem uses dynamically. The details of the resources are also uniquely provided in each subsystem and when subsystems are created, the size of necessary resource control blocks is specified.

Multiple resource control blocks can be created for one subsystem. Each resource control block is identified by the Resource ID. Moreover, the number of resource control blocks is equal in all subsystems. Resource control blocks with the same resource ID are grouped together and called resource groups.

When a resource group is created, resource control blocks are created with the same ID in each subsystem. When a resource group is deleted, the resource control block for each subsystem is also deleted. Moreover, when a subsystem is newly defined, resource control blocks for resource groups that already exist are created (Figure 23).

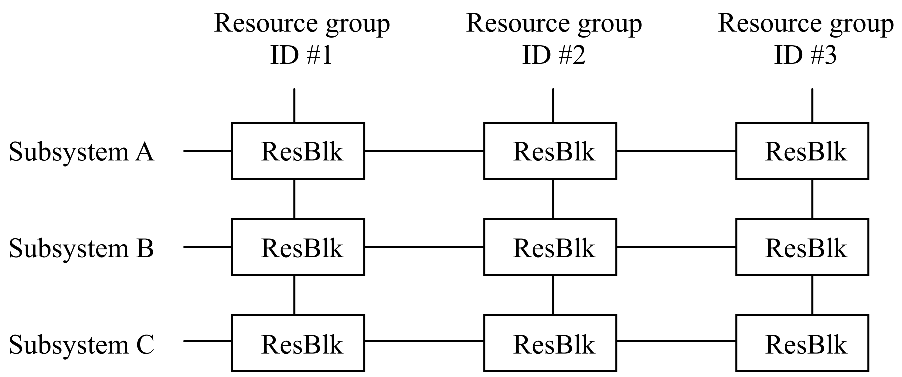


Figure 23 Relationship Between Subsystems and Resource Groups

System resource groups always exist as special resource groups. System resource groups are one or more resource groups which always exist when a system starts. System resource groups cannot be deleted. Except for existing at all times, there will be no parts which are different between the system resource group and other resource groups.

Tasks can specify the resource group to which they belong during creation. If the resource group is not specified, the task belongs to the system resource group.

[Additional Notes]

Intended uses of resource groups are described in the examples.

File management can be thought of as in the example on subsystems. This subsystem maintains various types of information on disks, directories, and files when they are accessed during serial operations.

When multiple tasks independently use the file management subsystem at the same time (in parallel), information stored by the subsystem must be held independently in each task. This is due to the reason that the files being accessed in each task are different, etc. In this case, information held by the subsystem is allocated in the resource control block and if a resource group is created when a task is created, information can be managed as independent resource control blocks in each task.

Situations in which a subsystem would like to manage information in each task can occur in addition to file control. If a resource group is created for each task, a resource control block for each task can be created in all subsystems. Moreover, if the resource group is deleted when the task is deleted, the resource control block for the task can be deleted in all subsystems.

If a certain subsystem does not need information management of each task, it can specify 0 for the size of the resource control block.

In reality, it is not necessary to individually create resource control blocks for all tasks and in most cases, it is sufficient that if a resource group is created by an independent program unit comprising the program in large application programs.

For example, in Standard Extension, applications are comprised of units called processes, and these processes are comprised of tasks for multiple T-Kernels. Standard Extension creates a resource group when a process is created, and tasks within the same process belong to the same resource group. When a process is deleted, the resource group is deleted.

Define Subsystem tk_def_ssy

tk_def_ssy: Define Sub-System

[C Language Interface]

```
ER ercd = tk_def_ssy ( ID ssid, T_DSSY *pk_dssy ) ;
```

[Parameters]

ID	ssid	Subsystem ID
T_DSSY*	pk_dssy	Subsystem definition information

pk_dssy detail:

ATR	ssyatr	Subsystem attributes
PRI	ssypri	Subsystem priority
FP	svchdr	Extended SVC handler address
FP	breakfn	Break function address
FP	startupfn	Startup function address
FP	cleanupfn	Cleanup function address
FP	eventfn	Event handling function address
INT	resblksz	Resource control block size (in bytes)

──(Other implementation-dependent parameters may be added beyond this point.))──

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (ssid is invalid or cannot be used)
E_NOMEM	Insufficient memory (memory for control block cannot be allocated)
E_RSATR	Reserved attribute (svcatr is invalid or cannot be used)
E_PAR	Parameter error (pk_dssy is invalid or cannot be used)
E_OBJ	ssid is already defined (when pk_dssy = NULL)
E_NOEXS	ssid is not defined (when pk_dssy = NULL)

[Description]

Defines subsystem ssid.

A subsystem ID must be assigned to each subsystem without overlapping with other subsystems. The OS does not have a function for assigning these automatically.

Subsystem IDs 1 to 9 are reserved for T-Kernel use. 10 to 255 are numbers used by middleware, etc. The maximum usable subsystem ID value is implementation-dependent and may be lower than 255 in some implementations.

ssyatr indicates system attributes in its low bits and implementation-dependent attributes in the high bits. The system attributes in ssyatr are not assigned in this version, and no system attributes are used.

ssypri indicates the subsystem priority. The startup function, cleanup function, and event handling function are called in order of priority. The order of calling when priority is the same is undefined. Subsystem priority 1 is the highest priority, with larger numbers indicating lower priorities. The range of priorities that can be specified is implementation-dependent, but it must be possible to assign at least priorities 1 to 16.

NULL can be specified in breakfn, startupfn, cleanupfn, and eventfn, in which case the corresponding function will not be called.

Specifying pk_dssy = NULL deletes a subsystem definition. The ssid subsystem resource control block will also be deleted.

Subsystems consist of the extended SVC handler to accept the extended SVC, each function (break function/start-up function/cleanup function/event function) for accepting requests by T-Kernel, and the resource control blocks. Each respective item is described here.

・Extended SVC handler

The extended SVC handler is the acceptance contact for requests from applications, etc. The extended SVC handler becomes subsystem API. It can be called in the same way as an ordinary system call, and is normally invoked using a trap instruction or the like.

The format of an extended SVC handler is as follows.

```
INT svchdr( VP pk_para, FN fncd )
{
        /*
                Branching by fncd
        */
        return retcode;    /* exit extended SVC handler */
}
```

fncd is a function code. The low 8 bits of the instruction code are the subsystem ID. The remaining high bits can be used in any way by the subsystem. Ordinarily they are used as a function code inside the subsystem. A function code must be a positive value, so the most significant bit is always 0.

pk_para points to a packet of parameters passed to this system call. The packet format can be decided by the subsystem. Generally a format like the stack passed to a C language function is used, which in many cases, is the same format as a C language structure.

The return code passed by an extended SVC handler is passed to the caller transparently as the function return value. As a rule, negative values are error codes and 0 or positive values are the return code for normal completion. If an extended SVC call fails for some reason, the OS error code (which is also a negative value) is returned to the caller without invoking the extended SVC handler, so it is best to avoid confusion with these values.

The format by which an extended SVC is called is dependent on the OS implementation. As a subsystem API, however, it must be specified in a C language function format independent of the OS implementation. The subsystem must provide an interface library for converting from the C language function format to the OS-dependent extended SVC calling format.

An extended SVC handler runs as a quasi-task portion.

It can be called from a task-independent portion, and in this case the extended SVC handler also runs as a task-independent portion.

・Break function

A break function is a function called when a task exception is raised for a task while an extended SVC handler is executing.

When a break function is called, the processing by the extended SVC handler running at the time the task exception was raised must be stopped promptly and control must be returned from the extended SVC handler to its caller. The processing for stopping the processing by the currently running extended SVC handler is called a break function.

The format of a break function is as follows.

```
void breakfn( ID tskid )
{
        /*
                Stop the running extended SVC handler
        */
}
```

tskid is the ID of the task where the task exception was raised.

A break function is called when a task exception is raised by tk_ras_tex. If extended SVC handler calls are nested, then when return is made from an extended SVC handler and the nesting level drops by 1, the extended SVC handler corresponding to the return destination is the one called.

A break function is called one time only for one extended SVC handler per one task exception.

If another nested extended SVC call is made while a task exception is raised, no break function is called for the called extended SVC handler.

A break function runs as a quasi-task portion. Its task context is either that of the task that called tk_ras_tex or that of the task where the task exception was raised (the task running an extended SVC handler). In the former case, the break function runs when tk_ras_tex is called, while in the latter case the break function runs when extended SVC nesting is reduced by one level. This means it is possible that the task executing the break function will be different from the task executing the extended SVC handler. In such a case, the break function and extended SVC handler run concurrently as controlled by task scheduling.

It is thus conceivable that the extended SVC handler will return to its caller before the break function finished executing, but in that case, the extended SVC handler waits at the point right before returning, until the break function completes. How this wait state maps to the task state transitions is implementation-dependent, but preferably it should remain in READY state (a READY state that does not go to RUN state). The precedence of a task may change while it is waiting for a break function to complete, but how task precedence is treated is implementation-dependent.

Similarly, an extended SVC handler cannot call an extended SVC until break function execution completes.

In other words, during the time from the raising of a task interrupt until the break function completes, the affected task must stay in the extended SVC handler that was executing at the time of the task exception.

If a break function and extended SVC handler run in different task contexts and the break function task priority is lower than the extended SVC handler task priority, the task priority of the break function is raised to the same priority as the extended SVC handler task only during the time while the break handler is executing. On the other hand, if the break function task priority is the same as or higher than that of the extended SVC handler, the priority does not change. The priority that gets changed is the current priority; the base priority stays the same.

The change in priority occurs only right before entry into the break function; any changes after that in the extended SVC handler task priority are not followed up by further changes in priority of the break function task. In no case does a change in the break function priority while a break function is running result in a priority change in the extended SVC handler task. At the same time there is no restriction on priority changes because a break function is running.

When the break function completes, the current priority of its task reverts to base priority. If a mutex was locked, however, the priority reverts to that as adjusted by the mutex. (In other words, the ability is provided to adjust the current priority at the entry and exit of the break function only; other than that, the priority is the same as when an ordinary task is running.)

・Startup function

A startup function is called by issuing the tk sta ssy system call. It performs resource control block initialization processing.

The format of a startup function is as follows.

```
void startupfn( ID resid, INT info )
{
        /*
                Resource control block initialization processing
        */
}
```

resid is the ID of the resource group to be initialized, and info is a parameter that can be used in any way. Both are passed to tk_sta_ssy.

Even if initialization of the resource control block fails for some reason, the startup function must be terminated normally. If the resource control block could not be initialized, then when an API (extended SVC) that cannot be executed normally as a result is called, error is passed in the return code of that API.

A startup function runs as a quasi-task portion in the context of the task that called tk_sta_ssy.

・Cleanup function

A cleanup function is called by issuing the tk cln ssy system call, and performs resource release processing.

The format of a cleanup function is as follows.

```
void cleanupfn( ID resid, INT info )
{
        /*
                Resource release processing
        */
}
```

resid is the ID of the resource group subject to resource release, while info is a parameter that can be used freely. Both are parameters passed to tk_cln_ssy.

Even if resource release fails for some reason, the cleanup function must be terminated normally. The error handling, such as logging of errors, can be decided for each subsystem.

After the cleanup function completes its processing, the resource control block is automatically cleared to 0. If no cleanup function was defined (cleanupfn = NULL), the tk_cln_ssy system call clears the resource control block to 0.

A cleanup function runs as a quasi-task portion in the context of the task that called tk_cln_ssy.

・Event handling function

An event handling function is called by issuing the tk evt ssy system call. It processes various requests made to a subsystem.

Note that it does not carry the obligation to process all requests for all subsystems. If processing is not required, it can simply return E_OK without performing any operation.

The format of an event handling function is as follows.

```
ER eventfn( INT evttyp, ID resid, INT info )
{
        /*
                Event processing
        */
        return ercd;
}
```

evttyp indicates the request type, resid gives the ID of the resource group, and info is a parameter that can be used freely. All these parameters are passed to tk_evt_ssy. If the system call is not invoked for any particular resource group, resid can be set to 0.

If processing completes normally, E_OK is passed in the return code; otherwise an error code (negative value) is returned.

The following event types evttyp are defined. See 5.3 for details.

```
#define TSEVT_SUSPEND_BEGIN     1       /* before suspending device */
#define TSEVT_SUSPEND_DONE      2       /* after suspending device */
#define TSEVT_RESUME_BEGIN      3       /* before resuming device */
#define TSEVT_RESUME_DONE       4       /* after resuming device */
#define TSEVT_DEVICE_REGIST     5       /* device registration notice */
#define TSEVT_DEVICE_DELETE     6       /* device deletion notice */
```

An event handling function runs as a quasi-task portion in the context of the task that called tk_evt_ssy.

・Resource control blocks

Resource control blocks are memory blocks for grouping resources and managing the blocks to which the resources belong. The size of the memory specified by resblksz is prepared for each resource group one at a time. If resblksz =0 is specified, a resource control block is not allocated. However, in this case the resource ID (refer to tk_cre_res) is allocated.

Tasks belong to one of the resource groups. When there is a request from a certain task for the subsystem, and the resources in the subsystem are allocated in the task, the allocation information is registered in the resource control block. Which resources are registered in the resource control block and how they are registered are decided by the subsystem side.

Since the OS is not concerned about the details of the resource control block, it can be freely used by the subsystem side. However, resblksz is made to be as small as possible. Therefore, if a larger memory block is necessary, the memory block must be prepared separately on the subsystem side and the address must be registered in the resource control block.

Resource control blocks are resident memory of system space.

[Additional Notes]

Extended SVC handlers as well as break functions, startup functions, cleanup functions and event handling functions all are equivalent of the TA_HLNG attribute only. There is no means of specifying the TA_ASM attribute.

Prior to initialization of a resource control block by a startup function, and after resource release by a cleanup function, the behavior if an extended SVC is called by a task belonging to that resource group is dependent on the subsystem implementation. The OS does not make any attempt to prevent this kind of call. Basically it is necessary to avoid calling an extended SVC before calling a startup function and after calling a cleanup function.

There may be cases where, for some reason or other, a break function, cleanup function or event handling function is called without first calling a startup function. These functions must execute normally even in such a case. A resource control block is cleared to 0 when it is first created and when cleanup processing is executed by tk_cln_ssy. Accordingly, even if it was not initialized properly by a startup function, the resource control block can still be assumed to have been cleared to 0.

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

Call Startup/Cleanup Function of Sub-System tk_sta_ssy tk_cln_ssy

tk_sta_ssy: Call StartUp Function of Sub-System
tk_cln_ssy: Call CleanUp Function of Sub-System

[C Language Interface]

```
ER ercd = tk_sta_ssy ( ID ssid, ID resid, INT info ) ;
ER ercd = tk_cln_ssy ( ID ssid, ID resid, INT info ) ;
```

[Parameters]

ID	ssid	Subsystem ID
ID	resid	Resource ID
INT	info	Any parameter

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (ssid or resid is invalid or cannot be used)
E_NOEXS	Object does not exist (the subsystem specified in ssid is not defined)
E_CTX	Context error (issued from task-independent portion or in dispatch disabled state)

[Description]

Calls the startup function/cleanup function of the subsystem specified in ssid.

Specifying ssid = 0 makes the system call applicable to all currently defined subsystems. In this case, the startup/cleanup functions of each subsystem are called in sequence.

tk_sta_ssy: Calls in order starting from the highest subsystem priority.
tk_cln_ssy: Calls in order starting from the lowest subsystem priority.

The order among subsystems having the same priority is not defined.

If there are dependency relationships among different subsystems, the subsystem priority must therefore be set with those relationships in mind. If, for example, subsystem B uses functions in subsystem A, then the priority of subsystem A must be set higher than that of subsystem B.

Even if these system calls are issued for a subsystem with no startup function or cleanup function defined, those functions are simply not called; no error results.

If during startup/cleanup function execution a task exception is raised for the task that called tk_sta_ssy or tk_cln_ssy, the task exception is held until the startup/cleanup function completes its processing.

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

Call Event Function of Sub-System tk_evt_ssy

tk_evt_ssy: Call Event Function of Sub-System

[C Language Interface]

```
ER ercd = tk_evt_ssy ( ID ssid, INT evttyp, ID resid, INT info ) ;
```

[Parameters]

ID	ssid	Subsystem ID
INT	evttyp	Event request type
ID	resid	Resource ID
INT	info	Any parameter

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (ssid, resid is invalid or cannot be used)
E_NOEXS	Object does not exist (the subsystem specified in ssid is not defined)
E_CTX	Context error (issued from task-independent portion or in dispatch disabled state)

Other Error code returned by the event handling function

[Description]

Calls the event handling function of the subsystem specified in ssid.

Specifying ssid = 0 makes the system call applicable to all currently defined subsystems. In this case, the event handling function of each subsystem is called in sequence.

When evttyp is an odd number: Calls in order starting from the highest subsystem priority.

When evttyp is an even number: Calls in order starting from the lowest subsystem priority.

The order among subsystems having the same priority is not defined.

If this system call is issued for a subsystem with no event handling function defined, the function is simply not called; no error results.

If this system call is not invoked for any particular resource group, resid = 0 is specified.

If the event handling function returns an error, the error code is passed transparently in the system call return code. When ssid = 0 and an event handler returns an error, the event handling functions of all other subsystems continue to be called. In the system call return code, only one error code is returned even if more than one event handling function returned an error. It is not possible to know which subsystem's event handling function returned the error.

If during event handling function execution a task exception is raised for the task that called tk_evt_ssy, the task exception is held until the event handling function completes its processing.

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

Reference Subsystem Status tk_ref_ssy

tk_ref_ssy: Refer Sub-System Status

[C Language Interface]

```
ER ercd = tk_ref_ssy ( ID ssid, T_RSSY *pk_rssy ) ;
```

[Parameters]

ID	ssid	Subsystem ID
T_RSSY*	pk_rssy	Subsystem definition information

[Return Parameters]

ER	ercd	ErrorCode	Error code

pk_rssy detail:

PRI	ssypri	Subsystem priority
INT	resblksz	Resource control block size (in bytes)

——(Other implementation-dependent parameters may be added beyond this point.)——

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (ssid is invalid or cannot be used)
E_NOEXS	Object does not exist (the subsystem specified in ssid is not defined)
E_PAR	Parameter error (pk_rssy is invalid or cannot be used)

[Description]

References information about the status of the subsystem specified in ssid.
resblksz returns the size of the resource control block specified with tk_def_ssy.
If the subsystem specified in ssid does not exist, E_NOEXS is returned.

[Items Concerning AMP T-Kernel]

There are no differences with the T-Kernel 1.00 Specification.

tk_cre_res: Create Resource Group

[C Language Interface]

ID resid = tk_cre_res (void) ;

[Parameters]

None

[Return Parameters]

ID	resid	Resource ID
	or	Error Code

[Error Codes]

E_LIMIT	Number of resource groups exceeds the system limit
E_NOMEM	Insufficient memory (memory for control block cannot be allocated)

[Description]

Creates a new resource group, assigning it to a resource control block and resource ID.

Resource IDs are assigned in common for the entire system. A separate resource control block is created for each subsystem.

In some cases, a new subsystem will be defined when a resource group is already created. Even in such a case, it is necessary to create a resource control block of an already existing resource group for the newly registered subsystem. In other words, there may be cases where resource control block creation must be performed by tk_def_ssy.

For example, if a new subsystem ID is defined in a situation like that shown in Figure 19, resource control blocks with resource IDs #1, #2, and #3 must automatically be created for the subsystem.

[Additional Notes]

A Resource ID is, in some cases, used also as a logical space ID (lsid). Resource IDs should therefore be assigned values that can be used directly as logical space IDs or that can easily be converted for use as logical space IDs.

Resource control block creation might be implemented in either of the following ways.

(A) At the time of subsystem definition (tk_def_ssy), create as many resource control blocks as the maximum number of resource groups, and use tk_cre_res simply to assign them.

(B) Use tk_cre_res to create as many resource control blocks as there are subsystems and assign them.

Since the specification requires clearing a resource control block to 0 when it is initially created, the timing of this clearing to 0 differs between methods (A) and (B). This difference should not have much of an effect; but since method (A) will have fewer cases of clearing to 0, subsystems must be implemented assuming (A). Method (A) is also recommended for the OS implementation.

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

tk_del_res: Delete Resource Group

[C Language Interface]

ER ercd = tk_del_res (ID resid) ;

[Parameters]

ID	resid	Resource ID

[Return Parameters]

ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (resid is invalid or cannot be used)
E_NOEXS	Object does not exist (the resource specified in resid does not exist)

[Description]

Deletes the resource control blocks of the resource group specified in resid, and releases the resource ID.

The resource control blocks of all subsystems are deleted.

[Additional Notes]

Resources are deleted even if there are still tasks belonging to a resource to be deleted. In principle, resource deletion must be performed after exit and deletion of all tasks belonging to the resources. The behavior is not guaranteed if a resource is deleted while a task belonging to that resource remains and is calling a subsystem (extended SVC). Likewise, the behavior is not guaranteed if a task belonging to a deleted resource calls a subsystem (extended SVC).

The timing for actual resource control block deletion is implementation-dependent. (See tk_cre_res.)

The system resource group cannot be deleted (error code E_ID is returned).

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

Get Resource Management Block tk_get_res

tk_get_res: Get Resource Management Block

[C Language Interface]

```
ER ercd = tk_get_res ( ID resid, ID ssid, VP *p_resblk ) ;
```

[Parameters]

ID	resid	Resource ID
ID	ssid	Subsystem ID D

[Return Parameters]

VP	resblk		Resource control block
ER	ercd	ErrorCode	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (resid or ssid is invalid or cannot be used)
E_NOEXS	Object does not exist (the resource specified in resid or ssid does not exist)
E_PAR	Parameter error (value that cannot be used in p_resblk)

[Description]
Gets the address of the resource control block of resource group resid for subsystem ssid.

[Additional Notes]
E_OK might be returned even if this system call is issued for a deleted resource ID. Whether or not error (E_NOEXS) is returned in this case is implementation-dependent.

[Items Concerning SMP T-Kernel]
There are no differences from the T-Kernel 1.00 Specification.

Chapter 5 SMP T-Kernel/SM Functions

Details of the functions provided by SMP T-Kernel System Manager (SMP T-Kernel/SM) are described in this chapter.
The following functions exist in SMP T-Kernel.

- System memory management functions
- Address space management functions
- Device management functions
- Interrupt management functions
- I/O port management functions
- Power-saving functions
- System configuration information management functions

[Overall notice and supplement]

- In principle, functions whose name are tk_- is extended SVC, others are library functions (including in-line functions) or macros of the C language.
- Some libraries and macros call some extended SVC or system calls indirectly.
- Error codes such as E_PAR, E_MACV, and E_NOMEM that always have the possibility of occurring are not described here unless there is some special reason for doing so.
- Except where otherwise noted, extended SVC and libraries of T-Kernel/SM can not be called from a task-independent portion and while dispatching and interrupts are disabled. There may be some limitations, however, imposed by particular implementations (E_CTX).
- Extended SVC and libraries of T-Kernel/SM can not be invoked from a lower protection level than that at which T-Kernel/OS system calls can be invoked (lower than TSVCLimit)(E_OACV).
- Extended SVC and libraries of T-Kernel/SM are reentrant except that the special explanation is given. But some functions make exclusive control internally.
- Detection of error codes E_PAR, E_MACV, and E_CTX is implementation-dependent; these may not always be detected as error. For this reason, the service calls must not be invoked in such a way that these errors might occur.

5.1 System Memory Management Functions

System memory management functions manage all the memory allocated dynamically by T-Kernel (system memory). This includes memory used internally by SMP T-Kernel as well as task stacks, message buffers, and memory pools.

System memory is managed in memory block units. The block size is normally the page size defined for the MMU.

A system that does not use an MMU can set any desired block size, but a size in the range of around 1 KB to 4 KB is recommended. Block size can be learned by calling tk_ref_smb.

System memory is allocated in the system space. SMP T-Kernel does not manage task space memory.

There are no differences between the system memory management functions of SMP T-Kernel and the T-Kernel 1.00 Specification.

5.1.1 System Memory Allocation

System memory management functions are called as extended SVC. They are for use not only in SMP T-Kernel but also in applications, subsystems and device drivers. In the case of SMP T-Kernel internal use, the calling of these functions without going through extended SVC calls is an implementation-dependent option.

The system calls for these functions are summarized in the table below. For more details refer to the explanation for each system call.

Call Name	Function	Different from T-Kernel 1.00 Specification
tk_get_smb	Allocate system memory	○
tk_rel_smb	Release system memory	○
tk_ref_smb	Acquires information regarding system memory	○

Different from T-Kernel 1.00 Specification ○:No ×:Yes △: Different only in that E_DACV is returned due to the access protection

tk_get_smb: Get System Memory Block

[C Language Interface]

```
ER ercd = tk_get_smb( VP *addr, INT nblk, UINT attr ) ;
```

[Parameters]

VP*	addr	Address to return the start address of the allocated memory area
INT	nblk	Block count of the allocated memory area
UINT	attr	Attribute of the allocated memory area

[Return Parameters]

ER	ercd	Error code
VP	addr	Start address of the allocated memory area

[Error Codes]

E_OK	Normal completion
E_NOMEM	Insufficient memory (memory is not allocated)
E_RSATR	Reserved attribute (attr is invalid or cannot be used)
E_PAR	Parameter error (addr is invalid)

[Description]

Allocates a memory space of a size accommodating the number of contiguous memory blocks specified in nblk, and having the attributes specified in attr. The start address of the allocated memory space is returned in addr.

```
attr := (TA_RNG0 || TA_RNG1 || TA_RNG2 || TA_RNG3) | [TA_NORESIDENT]

#define TA_NORESIDENT    0x00000010    /* nonresident */
#define TA_RNG0          0x00000000    /* protection level 0 */
#define TA_RNG1          0x00000100    /* protection level 1 */
#define TA_RNG2          0x00000200    /* protection level 2 */
#define TA_RNG3          0x00000300    /* protection level 3 */
```

The acquired memory does not belong to a resource group. If memory could not be allocated, E_NOMEM is returned.

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

Release System Memory tk_rel_smb

tk_rel_smb: Release System Memory Block

[C Language Interface]

```
ER ercd = tk_rel_smb( VP addr ) ;
```

[Parameters]

VP	addr	Address to return the start address of memory to be released

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_PAR	Parameter error (addr is invalid)

[Description]

Releases the memory specified by the start address addr. addr must be the address obtained by tk_get_smb.

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

Acquisition of System Memory Information tk_ref_smb

tk_ref_smb: Refer System Memory Block Information

[C Language Interface]

ER ercd = tk_ref_smb(T_RSMB *pk_rsmb) ;

[Parameters]

T_RSMB* pk_rsmb Packet address to return system memory information

[Return Parameters]

ER ercd Error code

pk_rsmb details
INT blksz block size (in bytes)
INT total total block count
INT free remaining free block count
/* Implementation-dependent information may be added beyond this point. */

[Error Codes]

E_OK Normal completion
E_PAR Parameter error (pk_rsmb is invalid)

[Description]

Gets information about system memory.

When virtual memory is used, there may be cases where the total block and remaining free block counts cannot be decided unequivocally. In such cases, the contents of total and free are implementation-dependent, but preferably they should be values such that free ÷ total gives a useful estimate of the remaining memory capacity.

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

5.1.2 Memory Allocation Libraries

Since system memory is allocated in block units, libraries are provided for dividing up those blocks for use.

```
void* Vmalloc ( size_t size )

void* Vcalloc ( size_t nmemb, size_t size )

void* Vrealloc ( void *ptr, size_t size )

void    Vfree ( void *ptr )
```

```
void* Kmalloc ( size_t size )

void* Kcalloc ( size_t nmemb, size_t size )

void* Krealloc ( void *ptr, size_t size )

void    Kfree ( void *ptr )
```

The functions are equivalent to the standard C libraries malloc, calloc, realloc, free and so on. V- means the function is for nonresident memory and K- for resident memory; in both cases the memory is assigned to the TSVCLimit protection level.

These functions cannot be called from a task-independent portion or while dispatching or interrupts are disabled. The behavior if they are called in those situations is undefined. (System failure is a possibility.)

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

5.2 Address Space Management Functions

The address space management function provides the library function for functions such as the setting and checks of address spaces, making resident/making non-resident (lock/unlock), memory cache operation, acquisition of physical addresses, and memory maps.

The address space management function is a low standard operation function that operates at a level close to hardware (refer to "2.10 Low Standard Operation Functions"). Therefore, this function largely depends on the hardware specification. Although a standard library function is provided in the address space management function, how far this function is realized is implementation-defined because it depends on the function of the hardware and system memory model.

5.2.1 Address Space Configuration

The address space setting sets the task space of tasks and access privilege information.
The access privilege to the memory of a task is decided according to the protection level. However, since the extended SVC is executed at protection level 0, if an extended SVC is called from the task, information on access privilege must be stored.

Memory access privilege is held as access privilege information for each task. Essentially access privilege information indicates the right to access at the protection level immediately before an extended SVC is called. If, for example, a task is running at protection level 3 when it calls an extended SVC, its access privilege information indicates the right to access at protection level 3.
Since the protection level when an extended SVC is executing is protection level 0, in the case of nested calling of an extended SVC from another extended SVC, access privilege information in the extended SVC for which a nested call was made indicates the right to access at protection level 0.
Memory access privilege information is set as follows.

- Immediately after a task is started, its access privilege is that specified when the task was created.
- When an extended SVC is called, the access privilege at the protection level at which it was running at the time of the call is set.
- Upon return from the extended SVC, the access privilege reverts to that at the time the extended SVC was called.
- Executing SetTaskSpace() copies the current access privilege of the target task to the invoking task.

ER SetTaskSpace (ID tskid)

Assigns to the invoking task the task space and access privilege information of the task specified in tskid. As a result, both the invoking task and target task have the same task space and access privilege information.
Note that this copying of task space information applies only at the time the function is called; if thereafter the task specified in tskid switches to a different address space and its access privilege changes, the invoking task is not affected by those changes (its address space and access privilege do not change accordingly). If the invoking task is calling an extended SVC, on return from the extended SVC its access privilege reverts to that prior to calling the extended SVC. Its task space, however, does not revert.
The task ID of the invoking task cannot be specified in tskid. However, if TSK_SELF is used to specify the invoking task, access privilege is set to the currently running protection level; task space is not switched in this case.

E_ID	tskid is invalid
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_OBJ	Invoking task specified by other than TSK_SELF

5.2.2 Address Space Checking

The following functions check whether access is allowed to the specified memory space, based on the current access privilege information. If access cannot be made (no privilege or the memory does not exist), they return error code E_MACV.

～R	Check for read access privilege.
～RW	Check for read and write access privilege.
～RE	Check for read and execute access privilege.

ER ChkSpaceR (VP addr, INT len)

ER ChkSpaceRW (VP addr, INT len)

ER ChkSpaceRE (VP addr, INT len)

Checks access privilege to the memory space of len bytes from location addr.

INT ChkSpaceBstrR (UB *str, INT max)

INT ChkSpaceBstrRW (UB *str, INT max)

Checks access privilege to the memory space from str up to the string termination ('¥0') or up to the number of characters (bytes) specified in max, whichever comes first. If max = 0 is set, max is ignored and privilege is checked up to the string termination.

If access is allowed, the length of the string (in bytes) is returned. If the string termination occurred up to the string length indicated in max, the length to the character before '¥0' is returned; if max characters occurred before the string termination, max is returned.

INT ChkSpaceTstrR (TC *str, INT max)

INT ChkSpaceTstrRW (TC *str, INT max)

```
typedef     UH        TC;         /* TRON character code */
#define     TNULL     ((TC)0)     /* TRON code string termination */
```

Checks access privilege to the memory space from str up to the TRON Code string termination (TNULL) or up to the number of characters (TC count) specified in max, whichever comes first. If max = 0 is set, max is ignored and privilege is checked up to the string termination.

If access is allowed, the length of the string (TC count) is returned. If the string termination occurred up to the string length indicated in max, the length to the character before TNULL is returned; if max characters occurred before the string termination, max is returned.

str must be an even-numbered address.

5.2.3 Lock Address Space

Address space is resident or nonresident memory. Generally memory is made resident or nonresident a page at a time and is managed in page units. For this reason, in many cases, the OS does not check for matching of locked and unlocked spaces. It is the responsibility of the calling side to make sure the same spaces are specified in lock and unlock operations.

ER LockSpace (VP addr, INT len)

Locks (makes resident) the memory space of len bytes from location addr.

E_MACV   The memory does not exist

ER UnlockSpace (VP addr, INT len)

Unlocks (makes nonresident) the memory space of len bytes from location addr. If the same space was locked more than once, it is not unlocked until the number of unlock operations equals the number of lock operations.

Note that it is not possible to unlock just part of a locked space.

[Additional Notes]

In systems without virtual memory, address spaces are not locked/unlocked, simply E_OK is returned. As a result, although the same source code can be used without depending on the existence of virtual memory in the system, actual operations will differ depending on the system.

5.2.3 Get Address Space Information

Gets various information on the address space.

```
ER ercd = GetSpaceInfo( VP addr, INT len, T_SPINFO *pk_spinfo )
```

addr Local address
len Memory space size (in bytes)

```
typedef struct t_spinfo{
    VP  paddr;        /* Physical address for addr */
    VP  page;         /* Physical address for the page to which addr belongs */
    INT pagesz;       /* Page size (in bytes) */
    INT cachesz;      /* Cache line size (in bytes) */
    INT cont;         /* Contiguous area size for the physical address from addr (in bytes) */
    /* Implementation-dependent information may be added beyond this point */
} T_SPINFO;
```

Acquires address space information on the space in len bytes from the logic address addr and returns in the return parameter pk_spinfo.

The corresponding physical address to addr is returned to paddr. The start physical address of the page to which addr belongs is returned in page.

The page size is returned in pagesz. Page size is the unit size when the cache mode is set by SetCacheMode.

The cache line size is returned in cachesz. Cache line size is the unit size when cache is controlled by ControlCache.

The size of the space where the physical address is contiguous from addr is returned in cont. Continuity is examined up to the len bytes. The len is 1 or more. When 0 or less is specified, the error code E_PAR is returned. The details of the return parameter pk_spinfo when an error occurs is indeterminate.

When a page out space exists in the specified area, memory space information directly before the area is returned. At this time, the return value is E_OK and the size of the area that acquired the information is returned in cont. When the start of the specified area is page out, the return value is E_OK and cont=0. Here the details of the return parameter pk_spinfo other than cont is indeterminate.

E_OK	Normal completion
E_PAR	Parameter error
E_MACV	Memory cannot be accessed; memory access privilege error

5.2.4 Cache Mode Setting

Sets the cache mode of the memory area. The setting of the cache mode is done in page units.

```
INT rlen = SetCacheMode( VP addr, INT len, UINT mode )
```

addr Start address
len Memory space size (in bytes)
mode Cache mode
Return code Size of the area the cache mode set (in bytes) or error code

The setting specified by mode is done from addr for the cache of the memory area of len bytes.

mode := (CM_OFF || CM_WB || CM_WT) | [CM_CONT]

CM_OFF	Cache off
CM_WB	Cache on (write-back)
CM_WT	Cache on (Write-through)
CM_CONT	Conducts cache setting only for the area the where physical address is contiguous

(Implementation-dependent mode may be added)

When CM_OFF is specified in mode, the cache mode is set to OFF after the cache is flushed (write back).
When CM_WT is specified in mode, after the cache is flushed the cache mode is set to write-through.

When CM_WB is specified in mode, the cache mode is set to write-back. At this time, whether or not the cache is flushed is system-dependent.

When CM_CONT is specified in mode, the cache mode is set only for the area where the physical address is contiguous from addr. When the physical address in the specified area is discontinuous or when a page out area exists, processing stops immediately prior to the address to be discontinued, and the size of the area where processing was completed is returned. When CM_CONT is not specified, the cache for the entire specified area is processed and the size of the area where processing was completed is returned.

Part of the cache mode or all settings may be disabled depending on the machine. When a disabled mode is specified, nothing is processed and E_NOSPT is returned.

The len is 1 or more. When 0 or less is specified, the error code E_PAR is returned.

The cache mode is set in page units. For this reason, since there is the possibility that cache access which is not intended may occur in the adjacent area, this must be noted for use. Page size is system-dependent and can be acquired by GetSpaceInfo.

E_PAR	Cache mode (mode), Memory space size (len) is invalid
E_NOSPT	Unsupported function

5.2.5 Control of Cache

The cache of the memory area can be controlled (flush/cancel). Cache control is done in cash line size units.

```
INT rlen = ControlCache( VP addr, INT len, UINT mode )
```

addr	Start address
len	Memory space size (in bytes)
mode	Control mode
Return code	Size of the area where cache was controlled (in bytes) or error

The control specified by mode is done from addr to the cache of len bytes memory area.

mode := (CC_FLUSH | CC_INVALIDATE) | [CC_SELF]

CC_FLUSH	Cache flush (Write-back)
CC_INVALIDATE	Cache invalidated
CC_SELF	Specifies own processor only

（Implementation-dependent mode may be added）

CC_FLUSH and CC_INVALIDATE are specifiable at the same time. In this case, after the cache is flushed, it is invalidated.

When processing succeeds, the size of the processed area is returned. When a page out area exists within the specified area, processing ends immediately prior to the area and the size of the area where processing was completed is returned.

When CC_SELF is specified, the specified cache control is executed by the processor which executed this call only. There is no influence on the cache of other processors. However, when the implementation of control for its own processor only is difficult according to the system, an E_NOSPT error is returned without implementing CC_SELF. When CC_SELF is not specified, cache control is executed for the cache of all processors.

Cache is controlled in cache line size units. For this reason, since there is the possibility that cache access which is not intended may occur in the adjacent area, this must be noted for use. Cache size is system-dependent and can be acquired by GetSpaceInfo.

E_PAR	Parameter error
E_NOSPT	Unsupported function

5.2.6 Get Physical Address

Gets the physical address of the memory for directly accessing memory from outside the CPU such as DMAC transfers. At the same time, the cache of the area whose physical address was acquired is set to OFF.

INT CnvPhysicalAddr (VP vaddr, INT len, VP *paddr)

vaddr	Logical address
len	Memory space size (in bytes)
paddr	Returns physical address
return code	Returns size (in bytes) of physical address contiguous space, or error

Gets the physical address corresponding to logical address vaddr, returning the result in paddr. This function also passes in the return code the size of contiguous space included in len bytes from vaddr. Accordingly, only the space of the size passed in the return code starting from paddr is valid.

The space for which the physical address is obtained must be locked (made resident).

On the assumption of DMA transfer, memory caching is turned off for the space whose physical address is obtained (the memory space starting from paddr of the size passed in the return code).When the space is made nonresident (unlocked), caching goes back on. If it is not possible to make memory cached off partly by a hardware limitation, this API flush the cache memory (that is, write it back and make it disable).

If this call is successful, the physical address paddr is the start address and directly accessing the memory in which the CPU such as DMAC transfers is not used is guaranteed in the space of the size passed in the return code.

E_MACV The memory does not exist

[Additional Notes]
In systems that operate by the physical address due to reasons such as not have an MMU, the address is not converted, and the value is returned as it is passed. However, it must be noted that memory cache is controlled.

5.2.7 Map Memory

Map memory functions secure the continuous memory area of the physical address on logical space.

ER MapMemory (VP paddr, INT len, UINT attr, VP *laddr)

Maps len bytes area which starts from paddr in the physical address to logical area returning the logical address in ***laddr**. The logical address is not to be specified, automatically allocated.

If paddr = NULL is specified, some continuous memory in physical memory is automatically allocated and mapped to logical area.

The mapped logical area has attributes specified with attr

attr := (MM_USER ｜｜ MM_SYSTEM) | [MM_READ] | [MM_WRITE] | [MM_EXECUTE] | [MM_CDIS]

MM_USER	User level access
MM_SYSTEM	System level access
MM_READ	Read access
MM_WRITE	Write access
MM_EXECUTE	Execution
MM_CDIS	Disable cache

These symbols differ from every machine. It requires these symbols to use. Some machines need other attributes except for these shows above.

The mapped memory area is made resident. It is not necessary to relock with LockSpace to make the area resident.

E_LIMIT	Insufficient logical area for mapping
E_NOMEM	Insufficient memory

ER UnmapMemory (VP laddr)

Unmaps logical area which allocated by MapMemory(). The logical address allocated by MapMemory() must be set to laddr. If some memory is allocated by MapMemory(), the memory is also unmapped.

[Additional Notes]

The map memory function is used when continuous physical memory areas are needed by the device.

When the area that is secured by specifying paddr=NULL is converted into the physical address with CnvPhysicalAddr(),the continuous physical memory area where direct access is possible and where the CPU is not used such as DMAC transfers memory can be acquired.

5.3 Device Management Functions

Various hardware are managed as devices. There are no differences between the device management functions of SMP T-Kernel and the T-Kernel 1.00 Specification.

5.3.1 Basic Concepts

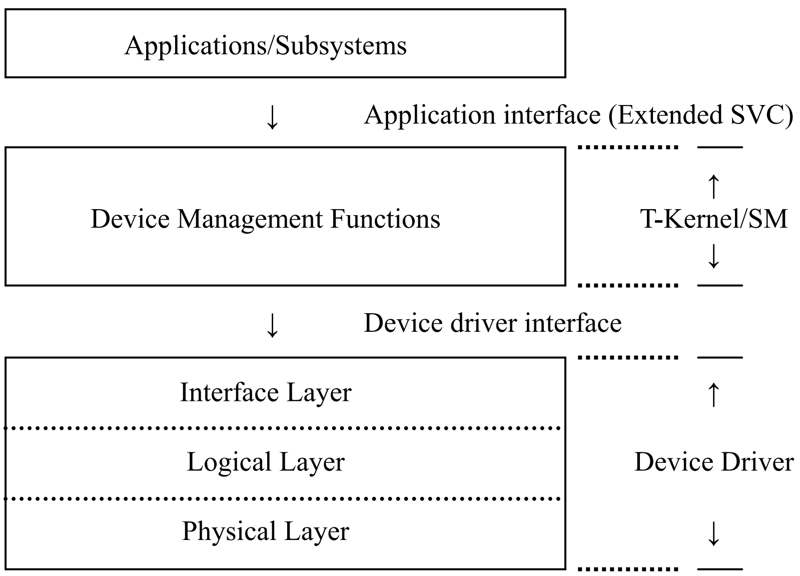


Figure 24: Device Management Functions

(1) Device Name (UB* type)

A device name is a string of up to 8 characters consisting of the following elements.

```
#define L_DEVNM 8      /* Device name length */
```

Type	Name indicating the device type Characters a to z and A to Z can be used.
Unit	One letter indicating a physical device Each unit is assigned a letter from a to z in order starting from a.
Subunit	One to three digits indicating a logical device Each subunit is assigned a number from 0 to 254 in order starting from 0.

Device names take the format type + unit + subunit. Some devices may not have a unit or subunit, in which case the corresponding field is omitted.

A name consisting of type + unit is called a physical device name. A name consisting of type + unit + subunit may be called a logical device name to distinguish it from a physical device name. If there is no subunit, the physical device name and logical device name are identical. The term “device name” by itself means the logical device name.

A subunit generally refers to a partition on a hard disk, but can be used to mean other logical devices as well.

Examples:	hda	Hard disk (entire disk)
	hda0	Hard disk (1st partition)
	fda	Floppy disk
	rsa	Serial port
	kbpd	Keyboard/pointing device

(2) Device ID (ID type)

By registering a device (device driver) with T-Kernel/SM, a device ID (> 0) is assigned to the device (physical device name). Device IDs are assigned to each physical device. The device ID of a logical device consists of the device ID assigned to the physical device to which is appended the subunit number + 1 (1 to 255).

devid: The device ID assigned at device registration

devid	Physical device
devid + n+1	The nth subunit (logical device)

Examples:	hda	devid	Entire hard disk
	hda0	devid + 1	1st partition of hard disk
	hda1	devid + 2	2nd partition of hard disk

(3) Device Attribute (ATR type)

Device attributes are defined as follows, in order to classify devices by their properties.

IIII IIII IIII IIII PRxx xxxx KKKK KKKK

The high 16 bits are device-dependent attributes defined for each device. The low 16 bits are standard attributes defined as follows.

```
#define TD_PROTECT          0x8000   /* P: write protection */
#define TD_REMOVABLE        0x4000   /* R: removable media */

#define TD_DEVKIND          0x00ff   /* K: device/media kind */
#define TD_DEVTYPE          0x00f0   /* device type */

                                     /* device type */
#define TDK_UNDEF           0x0000   /* undefined/unknown */
#define TDK_DISK            0x0010   /* disk device */

/* disk kind */
#define TDK_DISK_UNDEF      0x0010   /* miscellaneous disk */
#define TDK_DISK_RAM        0x0011   /* RAM disk (used as main memory) */
#define TDK_DISK_ROM        0x0012   /* ROM disk (used as main memory) */
#define TDK_DISK_FLA        0x0013   /* Flash ROM or other silicon disk */
#define TDK_DISK_FD         0x0014   /* floppy disk */
#define TDK_DISK_HD         0x0015   /* hard disk */
#define TDK_DISK_CDROM      0x0016   /* CD-ROM */
```

Currently no device types other than disks are defined. Other devices are assigned to undefined type (TDK_UNDEF). Note that device types are defined for the sake of distinguishing devices from the standpoint of the user as necessary, such as when applications must change their processing based on the type of device or media. Devices for which no such distinctions are necessary do not have to have a device type assigned.

See the individual device driver specifications regarding device-dependent attributes.

(4) Device Descriptor (ID type)

A device descriptor (> 0) is an identifier used for accessing a device, assigned by T-Kernel/SM when a device is opened.

A device descriptor belongs to a resource group. Operations using a device descriptor can be performed only by tasks belonging to the same resource group as the device descriptor. Error code (E_OACV) is returned for requests from tasks belonging to a different resource group.

(5) Request ID (ID type)

When an IO request is made to a device, a request ID (> 0) is assigned identifying the request. This ID can be used to wait for IO completion.

(6) Data Number (INT type)

Device data is specified by a data number. Data is classified into device-specific data and attribute data as follows.

Device-specific data: Data number >= 0

As device-specific data, the data numbers are defined separately for each device.

Examples

Disk	Data number = physical block number
Serial port	Data number = 0 only

Attribute data: Data number < 0
Attribute data specifies driver or device state acquisition and setting modes, and special functions, etc.
Data numbers common to devices are defined, but device-dependent attribute data can also be defined. Details are given later.

5.3.2 Application Interface

The functions below are provided as application interface functions, called as extended SVC. These functions cannot be called from a task-independent portion or while dispatch or interrupts are disabled (E_CTX).

```
ID   tk_opn_dev( UB *devnm, UINT omode )
ER   tk_cls_dev( ID dd, UINT option )
ID   tk_rea_dev( ID dd, INT start, VP buf, INT size, TMO tmout )
ER   tk_srea_dev( ID dd, INT start, VP buf, INT size, INT *asize )
ID   tk_wri_dev( ID dd, INT start, VP buf, INT size, TMO tmout )
ER   tk_swri_dev( ID dd, INT start, VP buf, INT size, INT *asize )
ID   tk_wai_dev( ID dd, ID reqid, INT *asize, ER *ioerr, TMO tmout )
INT  tk_sus_dev( UINT mode )
ID   tk_get_dev( ID devid, UB *devnm )
ID   tk_ref_dev( UB *devnm, T_RDEV *rdev )
ID   tk_oref_dev( ID dd, T_RDEV *rdev )
INT  tk_lst_dev( T_LDEV *ldev, INT start, INT ndev )
INT  tk_evt_dev( ID devid, INT evttyp, VP evtinf )
```

ID tk_opn_dev (UB *devnm, UINT omode)

devnm	Device name
omode	Open mode
return code	Device descriptor or error

Opens the device specified in devnm in the mode specified in omode, and prepares for device access. The device descriptor is passed in the return code.

```
omode := (TD_READ || TD_WRITE || TD_UPDATE) | [TD_EXCL || TD_WEXCL || TD_REXCL]
         | [TD_NOLOCK]

#define TD_READ      0x0001   /* read only */
#define TD_WRITE     0x0002   /* write only */
#define TD_UPDATE    0x0003   /* read/write */
#define TD_EXCL      0x0100   /* exclusive */
#define TD_WEXCL     0x0200   /* exclusive write */
#define TD_REXCL     0x0400   /* exclusive read */
#define TD_NOLOCK    0x1000   /* lock (making resident) not necessary */
```

TD_READ Read only
TD_WRITE Write only
TD_UPDATA Sets read and write access mode.
When TD_READ is set, tk_wri_dev() cannot be used.
When TD_WRITE is set, tk_rea_dev() cannot be used.

TD_EXCL Exclusive
TD_WEXCL Exclusive write
TD_REXCL Exclusive read
Sets the exclusive mode.
When TD_EXCL is set, all concurrent opening is prohibited.
When TD_WEXCL is set, concurrent opening in write mode (TD_WRITE or TD_UPDATE) is prohibited.
When TD_REXCL is set, concurrent opening in read mode (TD_READ or TD_UPDATE) is prohibited.

Table 7 Whether or not it is possible to open the same device at the same time

Present Open Mode		Concurrent Open Mode					
		No exclusive mode		TD_WEXCL		TD_EXCL	
		R	W	R	W	R	W
No exclusive mode	R	○	○	○	○	×	×
	W	○	○	×	×	×	×
TD_WEXCL	R	○	×	○	×	×	×
	W	○	×	×	×	×	×
TD_EXCL	R	×	×	×	×	×	×
	W	×	×	×	×	×	×

R = TD_READ   W = TD_WRITE or TD_UPDATE
○= Can be opened   × = Cannot be opened (E_BUSY)

TD_NOLOCK   Lock (making resident) not necessary

Indicates that a memory space (buf) specified in IO operations (tk_rea_dev and tk_wri_dev) has already been locked (made resident) on the calling side and does not have to be locked by the device driver. In this case, the device driver does not (must not) lock the area. This is used e.g. to perform disk access for page in/page out in a virtual memory system. Generally, it does not need to be specified.

The device descriptor belongs to the resource group of the task that opened the device.

When a physical device is opened, the logical devices belonging to it are all treated as having been opened in the same mode, and are processed as exclusive open.

E_BUSY   Device busy (exclusive open)
E_NOEXS   Device does not exist
E_LIMIT   Open count exceeds the limit
Other   Errors returned by device driver

ER tk_cls_dev (ID dd, UINT option)

dd   Device descriptor
option   Close option
return code   Error

Closes device descriptor dd. If a request is being processed, the processing is aborted and the device is closed.

option := [TD_EJECT]

```
#define TD_EJECT        0x0001        /* eject media */
```

TD_EJECT   Eject media

If the same device has not been opened by another task, the media is ejected. In the case of devices that cannot eject their media, the request is ignored.

The subsystem cleanup processing (tk_cln_ssy) closes all the device descriptors belonging to the resource group.

E_ID   dd is invalid or not open
Other   Errors returned by device driver

ID tk_rea_dev (ID dd, INT start, VP buf, INT size, TMO tmout)

dd   Device descriptor
start   Read start location (>= 0: Device-specific data, < 0: Attribute data)

buf	Buffer location for putting the read data
size	Read size
tmout	Request acceptance timeout (ms)
return code	Request ID or error

Starts reading device-specific data or attribute data from the specified device. This function starts reading only, returning to its caller without waiting for the read operation to finish. The space specified in buf must be retained until the read operation completes. Read completion is waited for by tk_wai_dev. The time required for read start processing differs with the device; return of control is not necessarily immediate.

In the case of device-specific data, the start and size units are decided for each device. With attribute data, start is an attribute data number and size is in bytes. The attribute data of the data number specified in start is read. Normally size must be at least as large as the size of the attribute data to be read. Reading of multiple attribute data in one operation is not possible. When size = 0 is specified, actual reading does not take place but the current size of data that can be read is checked.

Whether or not a new request can be accepted while a read or write operation is in progress depends on the device driver. If a new request cannot be accepted, the request is queued. The timeout for request waiting is set in tmout. The TMO_POL or TMO_FEVR attribute can be specified for tmout. Note that what times out is request acceptance. Once a request has been accepted, this function does not time out.

E_ID	dd is invalid or not open
E_OACV	Open mode is invalid (read not permitted)
E_LIMIT	Number of requests exceeds the limit
E_TMOUT	Busy processing other requests
E_ABORT	Processing aborted
Other	Errors returned by device driver

ER tk_srea_dev (ID dd, INT start, VP buf, INT size, INT *asize)

Synchronous read. This is equivalent to the following.

```
ER tk_srea_dev( ID dd, INT start, VP buf, INT size, INT *asize )
{
        ER    er, ioer;

        er = tk_rea_dev(dd, start, buf, size, TMO_FEVR);
        if ( er > 0 ) {
                er = tk_wai_dev(dd, er, asize, &ioer, TMO_FEVR);
                if ( er > 0 ) er = ioerr;
        }
        return er;
}
```

ID tk_wri_dev (ID dd, INT start, VP buf, INT size, TMO tmout)

dd	Device descriptor
start	write start location (>= 0: Device-specific data, < 0: Attribute data)
buf	Bu ffer holding data to be written
size	Size of data to be written
tmout	Request acceptance timeout (ms)
return code	Request ID or error

Starts writing device-specific data or attribute data to a device. This function starts writing only, returning to its caller without waiting for the write operation to finish. The space specified in buf must be retained until the write operation completes. Write completion is waited for by tk_wai_dev. The time required for write start processing differs with the device; return of control is not necessarily immediate.

In the case of device-specific data, the start and size units are decided for each device. With attribute data, start is an attribute data number and size is in bytes. The attribute data of the data number specified in start is written. Normally, size must be at least as large as the size of the attribute data to be written. Multiple attribute data cannot be written in one operation. When size = 0 is specified, actual writing does not take place but the current size of data that can be written is checked.

Whether or not a new request can be accepted while a read or write operation is in progress depends on the device driver. If a new request cannot be accepted, the request is queued. The timeout for request waiting is set in tmout. The TMO_POL or TMO_FEVR attribute can be specified for tmout. Note that what times out is request acceptance. Once a request has been accepted, this function does not time out.

E_ID	dd is invalid or not open
E_OACV	Open mode is invalid (write not permitted)
E_RONLY	Read-only device
E_LIMIT	Number of requests exceeds the limit
E_TMOUT	Busy processing other requests
E_ABORT	Processing aborted
Other	Errors returned by device driver

ER tk_swri_dev (ID dd, INT start, VP buf, INT size, INT *asize)

Synchronous write. This is equivalent to the following.

```
ER tk_swri_dev( ID dd, INT start, VP buf, INT size, INT *asize )
{
        ER    er, ioer;

        er = tk_wri_dev(dd, start, buf, size, TMO_FEVR);
        if ( er > 0 ) {
                er = tk_wai_dev(dd, er, asize, &ioer, TMO_FEVR);
                if ( er > 0 ) er = ioer;
        }

        return er;
}
```

ID tk_wai_dev (ID dd, ID reqid, INT *asize, ER *ioer, TMO tmout)

dd	Device descriptor
reqid	Request ID
asize	Returns the read/write data size
ioer	Returns IO error
tmout	Timeout (ms)
return code	Completed request ID or error

Waits for completion of request reqid for device dd. If reqid = 0 is set, this function waits for completion of any pending request to dd. This function waits for completion only of requests currently processing when the function is called. A request issued after tk_wai_dev was called is not waited for.

When multiple requests are being processed concurrently, the order of their completion is not necessarily the same as the order of request but is dependent on the device driver. Processing is, however, guaranteed to be performed in a sequence such that the result is consistent with the order of requesting. When processing a read operation from a disk, for example, the sequence might be changed as follows.

Block number request sequence 1 4 3 2 5
Block number processing sequence 1 2 3 4 5

Disk access can be made more efficient by changing the sequence as above with the aim of reducing seek time and spin wait time.

The timeout for waiting for completion is set in tmout. The TMO_POL or TMO_FEVR attribute can be specified for tmout. If a timeout error is returned (E_TMOUT), tk_wai_dev must be called again to wait for completion, since the request processing is ongoing. When reqid > 0 and tmout = TMO_FEVR are both set, the processing must be completed without timing out.

If the requested processing results in error (IO error, etc.) ioer is stored rather than a return code. The return code is used for errors when the request wait itself was not handled properly. When error is passed in the return code, ioer has no meaning. Note also that if error is passed in the return code, tk_wai_dev must be called again to wait for completion, since the request processing is ongoing.

If a task exception is raised during completion waiting by tk_wai_dev, the request in reqid is aborted and processing is completed. The result of aborting the requested processing is dependent on the device driver. When reqid = 0 was set, however, requests are not aborted but are treated as timeout. In this case, E_ABORT rather than E_TMOUT is returned.

It is not possible for multiple tasks to wait for completion of the same request ID at the same time. If there is a task waiting for request completion with reqid = 0 set, another task cannot wait for completion for the same device descriptor. Similarly, if there is a task waiting for request completion with reqid > 0 set, another task cannot wait for completion specifying reqid = 0.

E_ID	dd is invalid or not open reqid is invalid or not a request for dd
E_OBJ	Another task is already waiting for request reqid
E_NOEXS	No requests are being processed (only when reqid = 0)
E_TMOUT	Timeout (processing continues)
E_ABORT	Processing aborted
Other	Errors returned by device driver

INT tk_sus_dev (UINT mode)

Mode	Mode
return code	Suspend disable request count or error

Performs the processing specified in mode, then passes the resulting suspend disable request count in the return code.

mode := ((TD_SUSPEND | [TD_FORCE]) || TD_DISSUS || TD_ENASUS || TD_CHECK)

```
#define TD_SUSPEND     0x0001   /* suspend */
#define TD_DISSUS      0x0002   /* disable suspension */
#define TD_ENASUS      0x0003   /* enable suspension */
#define TD_CHECK       0x0004   /* get suspend disable request count */
#define TD_FORCE       0x8000   /* forcibly suspend */
```

TD_SUSPEND Suspend
If suspending is enabled, suspends processing.
If suspending is disabled, returns E_BUSY.

TD_SUSPEND|TD_FORCE Forcibly suspend
Suspends even in suspend disabled state.

TD_DISSUS Disable suspension
Disables suspension.

TD_ENASUS Enable suspension
Enables suspension.
If the enable request count is above the disable count for the resource group, no operation is performed.

TD_CHECK Get suspend disable count
Gets only the number of times suspend disable has been requested.

Suspension is performed in the following steps.

1. Processing prior to start of suspension in each subsystem
tk_evt_ssy(0, TSEVT_SUSPEND_BEGIN,0,0)
2. Suspension processing in non-disk devices
3. Suspension processing in disk devices
4. Processing after completion of suspension in each subsystem
tk_evt_ssy(0, TSEVT_SUSPEND_DONE,0,0)
5. SUSPEND state entered
tk_set_pow(TPW_DOSUSPEND)

Resumption from SUSPEND state is performed in the following steps.

1. Return from SUSPEND state
Return from tk_set_pow(TPW_DOSUSPEND)
2. Processing prior to start of resumption in each subsystem
tk_evt_ssy(0, TSEVT_RESUME_BEGIN, 0, 0)
3. Resumption processing in disk devices
4. Resumption processing in non-disk devices
5. Processing after completion of resumption in each subsystem
tk_evt_ssy(0,TSEVT_RESUME_DONE,0,0)

The number of suspend disable requests is counted. Suspension is enabled only if the same number of suspend enable requests is made. At system boot, the suspend disable count is 0 and suspension is enabled. There is only one suspend disable request count kept per system, but the system keeps track of the resource group making the request. It is not possible to clear suspend disable requests made in another resource group. When the cleanup function runs in a resource group, all the suspend requests made in that group are cleared and the suspend disable request count is reduced accordingly. The maximum suspend disable request count is implementation-dependent, but must be at least 255. When the upper limit is exceeded, E_QOVR is returned.

E_BUSY	Suspend already disabled
E_QOVR	Suspend disable request count limit exceeded

ID tk_get_dev (ID devid, UB *devnm)

devid	Device ID
devnm	Device name storage location
return code	Device ID of physical device or error

Gets the device name of the device specified in devid and puts the result in devnm.
devid is the device ID of either a physical device or a logical device.
If devid is a physical device, the physical device name is put in devnm.
If devid is a logical device, the logical device name is put in devnm.
devnm requires a space of L_DEVNM + 1 bytes or larger.

The device ID of the physical device to which device devid belongs is passed in the return code.

E_NOEXS	The device specified in devid does not exist

ID tk_ref_dev (UB *devnm, T_RDEV *rdev)

ID tk_oref_dev (ID dd, T_RDEV *rdev)

devnm	Device name
dd	Device descriptor
rdev	Device information
return code	Device ID or error

```
typedef struct t_rdev {
        ATR     devatr;     /* device attributes */
        INT     blksz;      /* block size of device-specific data (-1: unknown) */
        INT     nsub;       /* subunit count */
        INT     subno;      /* 0: physical device: 1 to nsub: subunit number+1 */
        /* Implementation-dependent information may be added beyond this point */
} T_RDEV;
```

Gets device information about the device specified in devnm or dd and puts the result in rdev. If rdev = NULL is set, the device information is not stored.

nsub indicates the number of physical device subunits belonging to the device specified in devnm or dd.

The device ID of the device specified in devnm is passed in the return code.

E_NOEXS The device specified in devnm does not exist.

INT tk_lst_dev (T_LDEV *ldev, INT start, INT ndev)

Ldev	Location of registered device information (array)
start	Starting number
ndev	Number to acquire
return code	Remaining device registration count or error

```
typedef struct t_ldev {
        ATR     devatr;                 /* device attributes */
        INT     blksz;                  /* device-specific data block size (-1: unknown) */
        INT     nsub;                   /* subunits */
        UB      devnm[L_DEVNM];         /* physical device name */
        /* Implementation-dependent information may be added beyond this point */
} T_LDEV;
```

Gets information about registered devices. Registered devices are managed per physical device. The registered device information is therefore also obtained per physical device.

When the number of registered devices is N, number are assigned serially to devices from 0 to N - 1. Starting from the number specified in start in accordance with this scheme, the number of registrations specified in ndev is acquired and put in ldev. The space specified in ldev must be large enough to hold ndev registration information. The number of remaining registrations after start (N - start) is passed in the return code.
If the number of registrations from start is fewer than ndev, all remaining registrations are stored. A value passed in return code less than or equal to (<=) ndev means all remaining registrations were obtained. Note that this numbering changes as devices are registered and deleted. For this reason, accurate information may not always be obtained if the acquisition is carried out over multiple operations.

E_NOEXS start exceeds the number of registered devices

INT tk_evt_dev (ID devid, INT evttyp, VP evtinf)

devid	Event destination device ID
evttyp	Driver request event type
evtinf	Information for each event type
return code	Return code from device driver or error

Sends a driver request event to the device (device driver) specified in devid.

The following driver request events are defined.

```
#define TDV_CARDEVT     1       /* PC Card event (see Card Manager) */
#define TDV_USBEVT      2       /* USB event (see USB Manager) */
```

The functioning of driver request events and the contents of evtinf are defined for each event type.

E_NOEXS	The device specified in devid does not exist
E_PAR	Internal device manager events (evttyp < 0) cannot be specified

5.3.3 Device Registration

The following device registration information is defined when registering a device. Device registration is performed for each physical device.

```
typedef struct t_ddev {
        VP    exinf;        /* extended information */
        ATR drvatr;         /* driver attributes */
        ATR devatr;         /* device attributes */
        INT nsub;           /* subunits */
        INT blksz;          /* block size of device-specific data (-1: unknown) */
        FP    openfn;       /* open count */
        FP    closefn;      /* close count */
        FP    execfn;       /* processing start function */
        FP    waitfn;       /* completion wait function */
        FP    abortfn;      /* abort processing function */
        FP    eventfn;      /* event function */
```

```
        /* Implementation-dependent information may be added beyond this point. */
} T_DDEV;
```

exinf is used to store any other desired information. The value of exinf is passed to the processing functions. Device management pays no attention to the contents.

drvatr sets device driver attribute information. The low bits indicate system attributes and the high bits are used for implementation-dependent attributes. The implementation-dependent attribute portion is used, for example, to indicate validity flags when implementation-dependent data is added to T_DDEV.

```
drvatr := [TDA_OPENREQ]
#define TDA_OPENREQ         0x0001    /* open/close each time */
```

TDA_OPENREQ

When a device is opened multiple times, normally openfn is called the first time it is opened and closefn the last time it is closed. If TDA_OPENREQ is specified, then openfn/closefn will be called for all open/close operations even in case of multiple openings.
Device attributes are specified in devatr. The details of device attribute settings are as noted above.
The number of subunits is set in nsub. If there are no subunits, 0 is specified.

blksz sets the block size of device-specific data in bytes. In the case of a disk device, this is the physical block size. It is set to 1 byte for a serial port, etc. For a device with no device-specific data it is set to 0. For an unformatted disk or other device whose block size is unknown, -1 is set. If blksz ≤ 0, device-specific data cannot be accessed. When device-specific data is accessed by tk_rea_dev or tk_wri_dev, size × blksz must be the size of the area being accessed, that is, the size of buf.
openfn, closefn, execfn, waitfn, abortfn, and eventfn set the entry address of processing functions. Details of the processing functions are discussed later.

ID tk_def_dev (UB *devnm, T_DDEV *ddev, T_IDEV *idev)

devnm	Physical device name
ddev	Device registration information
idev	Returns device initialization information
return code	Device ID or error

Registers a device with the device name set in devnm. If a device with device name devnm is already registered, the registration is updated with new information, in which case the device ID does not change. When ddev = NULL is specified, device devnm registration is deleted.
The device initialization information is returned in idev. This includes information set by default when the device driver is started, and can be used as necessary. When idev = NULL is set, device initialization information is not stored.

```
typedef struct t_idev {
    ID          evtmbfid;       /* event notification message buffer ID */
    /* Implementation-dependent information may be added beyond this point. */
} T_IDEV;
```

evtmbfid specifies the system default message buffer ID for event notification. If there is no system default event notification message buffer, 0 is set.
Notification like the following is made to each subsystem when a device is registered or deleted. devid is the device ID of the physical device registered or deleted.

Device registration or update:	tk_evt_ssy(0, TSEVT_DEVICE_REGIST, 0, devid)
Device deletion:	tk_evt_ssy(0, TSEVT_DEVICE_DELETE, 0, devid)

E_LIMIT	Number of registrations exceeds the system limit
E_NOEXS	The device specified in devnm does not exist (when ddev = NULL)

ER tk_ref_idv (T_IDEV *idev)

Idev	Returns device initialization information

return code Error

Gets device initialization information. The contents are the same as the information obtained by tk_dev_def().

5.3.4 Device Driver Interface

The device driver interface consists of processing functions specified when registering a device. These functions are called by device management and run as a quasi-task portion. They must be reentrant. The mutually exclusive calling of these processing functions is not guaranteed. If, for example, there are simultaneous requests from multiple devices for the same device, different tasks might call the same processing function at the same time. The device driver must apply mutual exclusion control in such cases as required.

IO requests to a device driver are made by means of the following request packet mapped to a request ID.

```
typedef struct t_devreq {
      struct t_devreq *next;      /* I: Link to request packet (NULL: termination) */
      VP          exinf;          /* X: Extended information */
      ID          devid;          /* I: Target device ID */
      INT         cmd:4;          /* I: Request command */
      BOOL        abort:1;        /* I: TRUE if abort request */
      BOOL        nolock:1;       /* I: TRUE if lock (making resident) not needed */
      INT         rsv:26;         /* I: reserved (always 0) */
      T_TSKSPC    tskspc;         /* I: Task space of requesting task */
      INT         start;          /* I: Starting data number */
      INT         size;           /* I: Request size */
      VP          buf;            /* I: IO buffer address */
      INT         asize;          /* O: Size of result */
      ER          error;          /* O: Error result */
      /* Implementation-dependent information may be added beyond this point. */
} T_DEVREQ;
```

I indicates an input parameter and O an output parameter. Input parameters must not be changed by the device driver. Parameters other than input parameters (I) are initially cleared to 0 by device management. After that, device management does not modify them.

next is used to link the request packet. In addition to being used for keeping track of request packets in device management, it is used also by the completion wait function (waitfn) and abort function (abortfn).

exinf can be used freely by the device driver to store any other information. Device management does not pay attention to its contents.

The device ID of the device to which the request is issued is specified in devid.

The request command is specified in cmd as follows.

```
cmd := (TDC_READ || TDC_WRITE)

#define TDC_READ    1    /* read request */
#define TDC_WRITE   2    /* write request */
```

If abort processing is to be carried out, abort is set to TRUE right before calling the abort function (abortfn). abort is a flag indicating whether abort processing was requested, and does not indicate that processing was aborted. In some cases, abort is set to TRUE even when the abort function (abortfn) is not called. Abort processing is performed when a request with abort set to TRUE is actually passed to the device driver.

nolock indicates that the memory space specified in buf has already been locked (made resident) and does not need to be locked by the device driver. In this case, the device driver must not lock the memory space. (nolock is specified when there is a possibility of wrong operation if the device driver performs a lock. Accordingly, when nolock = TRUE, the device driver must not lock the space.)

tskspc sets the task space of the requesting task. Since processing functions are called in the context of the requesting task, tskspc is the same as the task space of the processing function. If, however, the actual IO processing (read/write in the space specified in buf) is performed by a separate task in the device driver, it is necessary to switch the task space of the task performing the IO processing to the task space of the requesting task.

The start and size parameters for the tk_rea_dev or tk_wri_dev calls are copied to start and size in the request packet.

Similarly, the buf parameter for tk_rea_dev or tk_wri_dev is copied to buf in the request packet. The memory space specified in buf may be nonresident in some cases or task space in others. Care must therefore be taken regarding the following points.

- Nonresident memory cannot be accessed from a task-independent portion or while dispatching or interrupts are disabled.
- Task space memory cannot be accessed from another task space.

For these reasons, switching of task space or making memory space resident must be performed as necessary. Special attention is needed when access is made by an interrupt handler. Generally it is best not to access buf directly from an interrupt handler. Before accessing the buf memory space, the validity of buf must be checked using an address space check function (ChkSpace., described later below).

The device driver sets in asize the value returned in asize by tk_wai_dev.

The device driver sets in error the error code passed by tk_wai_dev in its return code. E_OK indicates a normal result.

Open Function: ER openfn(ID devid, UINT omode, VP exinf)

devid	Device ID of the device to open
omode	Open mode (same as tk_opn_dev)
exinf	Extended information set at device registration
return code	Error

The open function openfn is called when tk_opn_dev is invoked.

The function openfn performs processing to enable the use of a device. Details of the processing are device-dependent; if no processing is needed, it does nothing. The device driver does not need to remember whether a device is open or not, nor is it necessary to treat as error the calling of another processing function only because the device was not opened (openfn had not been called). If another processing function is called for a device that is not open, the necessary processing may be performed as long as there is no problem in device driver operation.

When openfn is used to perform operations such as device initialization in principle, no processing that causes a WAIT state should be performed. The processing and return from openfn must be as prompt as possible. In the case of a device such as a serial port for which it is necessary to set the communication mode, for example, the device can be initialized when the communication mode is set by tk_wri_dev. In such cases, there is no need for openfn to initialize the device.

When the same device is opened multiple times, normally this function is called only by the first time. If, however, the driver attribute TDA_OPENREQ is specified in device registration, this function is called each time the device is opened.

Since processing related to the mode of opening and the opening of device multiple times is handled by device management, processing related to them is not required in the openfn function. Similarly, omode is simply passed as reference information; no processing related to omode is required.

Close Function: ER closefn(ID devid, UINT option, VP exinf)

devid	Device ID of the device to close
option	Close option (same as tk_cls_dev)
exinf	Extended information set at device registration
return code	Error

The close function closefn is called when tk_cls_dev is invoked.

The closefn function performs processing to end use of a device. Details of the processing are device-dependent; if no processing is needed, it does nothing.

If the device is capable of ejecting media and TD_EJECT is set in option, media ejection is performed.

When closefn is used to perform device shutdown processing or media ejection, in principle no processing should be performed that causes a WAIT state. The processing and return from closefn must be as prompt as possible. If media ejection takes time, control may be returned from closefn without waiting for the ejection to complete.

When the same device is opened multiple times, normally this function is called only the last time it is closed. If, however, the driver attribute TDA_OPENREQ is specified in device registration, this function is called each time the device is closed. In this case, TD_EJECT is specified in option only for the last time.

Since processing related to the mode of opening and the opening of device multiple times is handled by device management, processing related to them is not required in the closefn function.

Processing Start Function: ER execfn(T_DEVREQ *devreq, TMO tmout, VP exinf)

devreq	Request packet
tmout	Request acceptance timeout (ms)
exinf	Extended information set at device registration
return code	Error

The execfn function is called when tk_rea_dev or tk_wri_dev is invoked and starts the processing requested in devreq. This function only starts the requested processing, returning to its caller without waiting for the processing to complete. The time required to start processing depends on the device driver; this function does not necessarily complete immediately.

When new processing cannot be accepted, this function goes to WAIT state for request acceptance. If the new request cannot be accepted within the time specified in tmout, the function times out. The attribute TMO_POL or TMO_FEVR can be specified in tmout. If the function times out, E_TMOUT is passed in the execfn return code. The timeout applies only to request acceptance and not to the processing after acceptance.

When error is passed in the execfn return code, the request is considered not to have been accepted and the request packet is discarded.

If processing is aborted before the request is accepted (before the requested processing starts), E_ABORT is passed in the execfn return code and the request packet is discarded. If processing is aborted after the processing has been accepted, E_OK is returned for this function. The request packet is not discarded until waitfn is executed and processing completes.

When processing is aborted, the important thing is to return from execfn as quickly as possible. If processing will end soon anyway without aborting, it is not necessary to abort.

Completion Wait Function: INT waitfn(T_DEVREQ *devreq, INT nreq, TMO tmout, VP exinf)

devreq	Request packet list
nreq	Request packet count
tmout	Timeout (ms)
exinf	Extended information set at device registration
return code	Completed request packet number or error

The waitfn function is called when tk_wai_dev is invoked.

devreq is a list of request packets in a chain linked by devreq->next. This function waits for completion of any of the nreq request packets starting from devreq. The final next is not necessarily NULL, so the count passed in nreq must always be observed. The number of the completed request packets (which one after devreq) is passed in the return code. The first one is numbered 0 and the last one is numbered nreq - 1. Here completion means any of normal completion, abnormal (error) termination, or abort.

The time to wait until completion is set in tmout. TMO_POL or TMO_FEVR can be specified as the tmout attribute. If the wait times out, the requested processing continues. The waitfn return code in case of timeout is E_TMOUT. The error parameter of the request packet does not change. Note that if return from waitfn occurs while the requested processing continues, error must be returned in the waitfn return code but the processing must be completed even when error is passed in the return code, and a value other than error must not be returned if processing is ongoing. As long as error is not passed in the waitfn return code, the request is considered to be pending and no request packet is discarded. When the number of a request packet whose processing was completed is passed in the waitfn return code, the processing of that request is considered to be completed and that request packet is discarded.

IO error and other device-related errors are stored in the error parameter of the request packet. Error is passed in the waitfn return code when completion waiting did not take place properly. The waitfn return code is set in the tk_wai_dev return code, whereas the request packet error value is returned in ioer.

Abort processing differs depending on whether the wait is for completion of a single request (nreq = 1) or multiple requests (nreq > 1). When completion of a single request is being waited for, the request currently being processed is aborted. When waiting for completion of multiple requests, only the wait is aborted (wait release), not the requested processing itself. When a wait for multiple requests is aborted (wait release), E_ABORT is passed in the waitfn return code.

During a wait for request completion, an abort request may be set in the abort parameter of a request packet. In such a case, if it is a single request, the request abort processing must be performed. If the wait is for multiple requests, it is also preferable that abort processing be executed, but it is also possible to ignore the abort flag.

When abort occurs, the important thing is to return from waitfn as quickly as possible. If processing will end soon anyway without aborting, it is not necessary to abort.

As a rule, E_ABORT is returned in the request packet error parameter when processing is aborted; but a different error code may be returned as appropriate based on the device properties. Returning E_OK on the basis that the processing right up to the abort is valid is also allowed. If processing completes normally to the end, E_OK is returned even if there was an abort request.

Abort Function: ER abortfn(ID tskid, T_DEVREQ *devreq, INT nreq, VP exinf)

tskid	Task ID of the task executing execfn or waitfn
devreq	Request packet list
nreq	Request packet count
exinf	Extended information set at device registration
return code	Error

The function abortfn causes execfn or waitfn to return promptly when the specified request is being executed. Normally this means the request being processed is aborted. If, however, the processing can be completed soon without aborting, it may not have to be aborted. The important thing is to return as quickly as possible from execfn or waitfn.

tskid indicates the task executing the request specified in devreq. In other words, it is the task executing execfn or waitfn. devreq and nreq are the same as the parameters that were passed to execfn or waitfn. In the case of execfn, nreq is always 1.

abortfn is called by a different task from the one executing execfn or waitfn. Since both tasks run concurrently, mutual exclusion control must be performed as necessary. It is possible that the abortfn function will be called immediately before calling execfn or waitfn, or during return from these functions. Measures must be taken to ensure proper operation in such cases. Before abortfn is called, the abort flag in the request packet whose processing is to be aborted is set to TRUE, enabling execfn or waitfn to know whether there is going to be an abort request. Note also that abortfn can make use of tk_dis_wai for any object.

When waitfn is executing for multiple requests (nreq > 1), this is treated as a special case differing as follows from other cases.

- Only the completion wait is aborted (wait release), not the requested processing.
- The abort flag is not set in the request packet (remains as abort = FALSE).

Aborting a request when execfn and waitfn are not executing is done not by calling abortfn but by setting the abort flag in the request packet. If execfn is called when the abort flag is set, the request is not accepted. If waitfn is called, abort processing is the same as that when abortfn is called.

If a request for which processing was started by execfn is aborted before waitfn was called to wait for its completion, the completion of the aborted processing is notified when waitfn is called. Even though processing was aborted, the request itself is not discarded until its completion has been confirmed by waitfn.

abortfn only starts abort processing, returning promptly without waiting for the abort to complete.

abortfn is called in the following cases.

- When a break function is executing after a task exception and the task that raised the exception requests abort processing, abortfn is used to abort the request being processed by that task.
- When a device is being closed by tk_cls_dev and by subsystem cleanup processing, and a device descriptor was processing a request, abortfn is used to abort the request being processed by that device descriptor.

Event Handling Function: INT eventfn(INT evttyp, VP evtinf, VP exinf)

evttyp	Driver request event type
evtinf	Information for each event type
exinf	Extended information set at device registration
return code	Return code defined for each event type or error

The following driver request event types are defined. Those with positive values are called by tk_evt_dev, and those with negative values are called inside device management.

```
#define TDV_SUSPEND     (-1)    /* suspend */
#define TDV_RESUME      (-2)    /* resume */
#define TDV_CARDEVT       1     /*   PC Card event (see Card Manager) */
#define TDV_USBEVT        2     /* USB event (see USB Manager) */
```

The processing performed by an event function is defined for each event type. Suspend and resume processing are discussed later below.

When a device event is called by tk_evt_dev, the eventfn return code is set as the tk_evt_dev return code.

Requests to event functions must be accepted even if another request is processing and must be processed as quickly as possible.

5.3.5 Attribute Data

Attribute data is classified broadly into the following three kinds of data.

- **Common attributes**
  Attributes defined in common for all devices (device drivers).
- **Device kind attributes**
  Attributes defined in common for devices (device drivers) of the same kind.
- **Device-specific attributes**
  Attributes defined independently for each device (device driver).

For the device kind attributes and device-specific attributes, see the specifications for each device. Only the common attributes are defined here.

Common attributes are assigned attribute data numbers in the range from -1 to -99. While common attribute data numbers are the same for all devices, not all devices necessarily support all common attributes. If an unsupported data number is specified, error code E_PAR is returned.

```
#define TDN_EVENT      (-1)   /* RW: event notification message buffer ID */
#define TDN_DISKINFO   (-2)   /* R-: disk information */
#define TDN_DISPSPEC   (-3)   /* R-: display device specification */
```

RW: read (tk_rea_dev)/write (tk_wri_dev) enabled
R-: read (tk_rea_dev) only

TDN_EVENT : Event Notification Message Buffer ID

Data type ID

The ID of the message buffer used for device event notification. Since the system default message buffer ID is passed in device registration, that ID is set as the initial setting when a driver is started.

If 0 is set, device events are not notified. Device event notification is discussed later below.

TDN_DISKINFO : Disk Information

Data type: DiskInfo

```
typedef enum {
    DiskFmt_STD      = 0,    /* standard (HD, etc.) */
    DiskFmt_2DD      = 1,    /* 2DD 720KB */
    DiskFmt_2HD      = 2,    /* 2HD 1.44MB */
    DiskFmt_CDROM    = 4,    /* CD-ROM 640MB */
} DiskFormat;

typedef struct {
    DiskFormat format;           /* format */
    UW      protect:1;           /* protected status */
    UW      removable:1;         /* removable */
    UW      rsv:30;              /* reserved (always 0) */
    W       blocksize;           /* block size in bytes */
    W       blockcount;          /* total block count */
} DiskInfo;
```

See the disk driver specification for details.

TDN_DISPSPEC: Display Device Specification

Data type: DEV_SPEC

```
typedef struct {
    H    attr;          /* device attributes */
    H    planes;        /* number of planes */
    H    pixbits;       /* pixel bits (boundary/valid) */
    H    hpixels;       /* horizontal pixels */
    H    vpixels;       /* vertical pixels */
    H    hres;          /* horizontal resolution */
    H    vres;          /* vertical resolution */
    H    color[4];      /* color information */
    H    resv[6];       /* reserved */
} DEV_SPEC;
```

See the screen driver specification for details.

5.3.6 Device Event Notification

A device driver sends events occurring in devices to the event notification message buffer (TDN_EVENT) as device event notification. The system default event notification message buffer is specified at the time of device registration, but can be changed later. The system default event notification message buffer is defined in TDEvtMbfSz in system configuration information.

The following event types are defined.

```
typedef enum tdevttyp {
      TDE_unknown          = 0,          /* undefined */
      TDE_MOUNT            = 0x01,       /* media mounted */
      TDE_EJECT            = 0x02,       /* media ejected */
      TDE_ILLMOUNT         = 0x03,       /* media illegally mounted */
      TDE_ILLEJECT         = 0x04,       /* media illegally ejected */
      TDE_REMOUNT          = 0x05,       /* media remounted */
      TDE_CARDBATLOW       = 0x06,       /* card battery alarm */
      TDE_CARDBATFAIL      = 0x07,       /* card battery failure */
      TDE_REQEJECT         = 0x08,       /* media eject request */
      TDE_PDBUT            = 0x11,       /* PD button state change */
      TDE_PDMOVE           = 0x12,       /* PD move */
      TDE_PDSTATE          = 0x13,       /* PD state change */
      TDE_PDEXT            = 0x14,       /* PD extended event */
      TDE_KEYDOWN          = 0x21,       /* key down */
      TDE_KEYUP            = 0x22,       /* key up */
      TDE_KEYMETA          = 0x23,       /* meta key state change */
      TDE_POWEROFF         = 0x31,       /* power switch off */
      TDE_POWERLOW         = 0x32,       /* low power alarm */
      TDE_POWERFAIL        = 0x33,       /* power failure */
      TDE_POWERSUS         = 0x34,       /* auto suspend */
      TDE_POWERUPTM        = 0x35,       /* clock update */
      TDE_CKPWON           = 0x41        /* auto power on notification */
} TDEvtTyp;
```

Device events are notified in the following format. The contents of event notification and size differ with each event type.

```
typedef struct t_devevt {
      TDEvtTyp         evttyp; /* event type */
      /* Information specific to each event type is appended here. */
} T_DEVEVT;
```

The format of device event notification with device ID is as follows.

```
typedef struct t_devevt_id {
      TDEvtTyp         evttyp;        /* event type */
      ID               devid;         /* device ID */
      /* Information specific to each event type is appended here. */
} T_DEVEVT_ID;
```

See the device driver specifications for event details.

Measures must be taken so that if event notification cannot be sent because the message buffer is full, the lack of notification will not adversely affect operation on the receiving end. One option is to hold the notification until space becomes available in the message buffer, but in that case, other device driver processing should not, as a rule, be allowed to fall behind as a result. Processing on the receiving end should be designed to the extent possible to avoid message buffer overflow.

5.3.7 Device Suspend/Resume Processing

Device drivers suspend and resume device operations in response to the issuing of suspend/resume

(TDV_SUSPEND/TDV_RESUME) events to the event handling function (eventfn). Suspend and resume events are issued only to physical devices.

Suspend Device (TDV_SUSPEND)

evttyp = TDV_SUSPEND
evtinf = NULL (none)

Suspension processing takes place in the following steps.

1. If there is a request being processed at the time, the device driver waits for it to complete, pauses it or aborts. Which of these options to take depends on the device driver implementation. However, since the suspension must be effected as quickly as possible, pause or abort should be chosen if completion of the request will take time.
   Suspend events can be issued only for physical devices, but the same processing is applied to all logical devices included in the physical device.
   Pause: Processing is suspended, then continues after the device resumes operation. **Abort:** Processing is aborted just as when the abort function (abortfn) is executed, and is not continued after the device resumes operation.
2. New requests other than a resume event are not accepted.
3. The device power is cut and other suspension processing is performed.

Abort should be avoided, if at all possible, because of its effects on applications. It should be used only in such cases as long input waits from a serial port, or when interruption would be difficult. Normally it is best to wait for completion of a request or, if possible, to pause (suspension and resumption).

Requests arriving at the device driver in suspend state are made to wait until operation resumes, after which acceptance processing is performed. If the request does not involve access to the device, however, or can otherwise be processed even during suspension, a request may be accepted without waiting for resumption.

Resume Device (TDV_RESUME)

evttyp = TDV_RESUME
evtinf = NULL (none)

Resumption processing takes place as follows.
1. The device power is turned back on, the device states are restored and other device resumption processing is performed.
2. Paused processing is resumed.
3. Request acceptance is resumed.

5.3.8 Special Properties of Disk Devices

A disk device has a special role to play in a virtual memory system. In order to realize virtual memory, the OS must call the disk driver for transferring data between memory and disk.

The need for the OS to perform data transfer with a disk arises when access is made to nonresident memory and the memory contents must be read from the disk (page in). The OS calls the disk driver in this case.

If nonresident memory is accessed in the disk driver, the OS must likewise call the disk driver. In such a case, if the disk driver treats the access to nonresident memory as a wait for page in, it is possible that the OS will again request disk access. Even then, the disk driver must be able to execute the later OS request.

A similar case may arise in suspension processing. When access is made to nonresident memory during suspension processing and a disk driver is called, if that disk driver is already suspended, page in will not be possible. To avoid such a situation, suspension processing should suspend other devices before disk devices. If there are multiple disk devices, however, the order of their suspension is indeterminate. For this reason, during suspension processing a disk driver must not access nonresident memory.

Because of the above limitations, a disk driver must not use (access) nonresident memory. It is possible, however, that the IO buffer (buf) space specified with tk_rea_dev or tk_wri_dev will be nonresident memory, since this is a memory location specified by the caller. In the case of IO buffers, therefore, it is necessary to make the memory space resident (see LockSpace) at the time of IO access.

5.4 Interrupt Management Functions

Interrupt management functions control the external interrupt flag and the interrupt controller of a CPU. Interrupt management functions are low level operation functions that operate the interrupt functions of hardware (refer to "2.10 Low Level Operation Functions"). Therefore, these functions largely depend on the function of hardware.

The following are given as standard specifications, but it may not be possible to follow these exactly on all systems. Implementors should strive to comply with these specifications to the extent possible; but where implementation is not feasible, full compliance is not mandatory. If functions not in the standard specifications are added, however, the function names must be different from those given here. In any case, DI(), EI(), and is DI() must be implemented in accordance with the standard specifications.

Interrupt management functions are provided as library functions or C language macros. These can be called from a task-independent portion and while dispatching and interrupts are disabled.

There are no differences between the specification for the interrupt management function of SMP T-Kernel and the T-Kernel 1.00 Specification. However, in general, the interrupt between processors is used in the internal processing of SMP T-Kernel. It must be noted that operation of the kernel is affected if interrupts between processors are prohibited by the prohibition all interrupts, etc.

5.4.1 CPU Interrupt Control

These functions are for CPU external interrupt flag control. Generally they do not perform any operation on the interrupt controller.

In SMP T-Kernel, the external interrupt flag of the processor that executes DI() and EI() is the object of control. The disable interrupts status of other processors does not change. In other words, the disable interrupts status exists independently in each processor.

DI() and EI() are C language macros.

DI (UINT intsts)

intsts CPU interrupt status (details are implementation-dependent) *This is not a pointer.

Disables all external interrupts. The status prior to disabling interrupts is stored in intsts.

EI (UINT intsts)

intsts CPU interrupt status (details are implementation-dependent)

Enables all external interrupts. More precisely, this macro restores the status in intsts. That is, the interrupt status reverts to what it was before interrupts were disabled by DI(). If there were interrupts disabled at the time DI() was executed, those interrupts are not enabled by EI(). All interrupts can be enabled, however, by specifying 0 in intsts.

intsts must be either the values stored in it by DI() or 0. If any other value is specified, the behavior is not guaranteed.

The value stored in DI() must be returned to the same processor by EI().In processors in disable interrupts status, if DI() and EI() are used as a pair, inevitably the value stored in the same processor is returned because the running program is not switched

BOOL isDI (UINT intsts)

intsts CPU interrupt status (details are implementation-dependent)
Return code: TRUE(not 0): Interrupts disabled FALSE: Interrupts enabled

Gets the status of external interrupt disabling stored in intsts. Interrupts disabled status is the status in which T-Kernel/OS determines that interrupts are disabled.

intsts must by the value stored by DI(). If any other value is specified, the behavior is not guaranteed.

Sample usage:

```
void foo()
{
        UINTintsts;

        DI(intsts);

        if ( isDI(intsts) ) {
```

```
        /* Interrupts were already disabled at the time this function was called. */
        } else {
        /* Interrupts were enabled at the time this function was called. */
        }

        EI(intsts);
}
```

[Items Concerning SMP T-Kernel]

In the internal processing of SMP T-Kernel, generally, interrupts between processors are used for communication and coordination between processors. When all external interrupts are prohibited by DI(), interrupts between these processors are prohibited and it is possible that the operation of SMP T-Kernel is obstructed. Therefore, DI() in applications must not be used as a rule. If there is a desire to prohibit a specific interrupt, use the Control of Interrupt Controller in the next paragraph without using DI() to prohibit target interrupts only.

Moreover, in past single processor programs, DI() may have been used for the purpose of exclusive control but exclusive control between processors cannot be conducted with multiprocessors. However, if software with high portability which includes multiprocessor systems is the aim, exclusive control should use the kernel object and the use of DI() must be limited to necessary minimum low level operations.

5.4.2 Control of Interrupt Controller

These functions control the interrupt controller. Generally they do not perform any operation with respect to the CPU interrupt flag.

```
typedef UINT    INTVEC;    /* interrupt vector */
```

The specific details of the interrupt vectors (INTVEC) are implementation-dependent. Preferably, however, they should be the same numbers as the interrupt definition numbers specified with tk_def_int, or should allow for simple conversion to and from those numbers.

UINT DINTNO(INTVEC intvec)

Converts an interrupt vector to the corresponding interrupt definition number.

void EnableInt(INTVEC intvec)

void EnableInt(INTVEC intvec, INT level)

In a system that allows interrupt priority level to be specified, the level parameter can be used to specify the interrupt priority level. The precise meaning of level is implementation-dependent.

Both methods with and without level must be provided.

void DisableInt(INTVEC intvec)

Disables the interrupt specified in intvec. Generally, interrupts raised while interrupts are disabled are made pending, and are raised after interrupts are enabled by EnableInt(). ClearInt() must be used if it is desired to clear interrupts occurring in interrupts disabled state.

void ClearInt(INTVEC intvec)

Clears any interrupts raised for intvec.

void EndOfInt (INTVEC intvec)

Issues EOI (End Of Interrupt) to the interrupt controller intvec must be an interrupt for which EOI can be issued. Generally this must be executed at the end of an interrupt handler.

BOOL CheckInt (INTVEC intvec)

Checks whether interrupt intvec has been raised. If interrupt intvec has been raised, it returns TRUE (value other than 0), else FALSE.

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

5.5 IO Port Access Support Functions

I/O port access support functions are low level operation functions to access I/O ports (refer to "2.10 Low Level Operation Functions"). Library function or C language macros are provided to I/O port access support functions and these can be called from the task independent portion and dispatch disable and interrupt prohibition states.

There are no differences between the I/O port access functions of SMP T-Kernel and the specifications in the T-Kernel 1.00 Specification.

5.5.1 IO Port Access

In a system with separate IO space and memory space, an IO port access function accesses IO space. In a system with memory-mapped IO only, an IO port access function accesses memory space. Using these functions will improve software portability and readability even in a memory-mapped IO system.

～_w Word (32-bit) units
～_h Half-word (16-bit) units
～_b Byte (8-bit) units

```
void out_w ( INT port, UW data )
```

```
void out_h ( INT port, UH data )
```

```
void out_b ( INT port, UB data )
```

port	IO port address
data	Data to be written

Writes data to an IO port.

```
UW in_w ( INT port )
```

```
UH in_h ( INT port )
```

```
UB in_b ( INT port )
```

port	IO port address
return code	Data to be read

Reads data from an IO port.

5.5.2 Micro wait

```
void WaitUsec ( UINT usec )
```

```
void WaitNsec ( UINT nsec )
```

usec Wait time (microseconds)
nsec Wait time (nanoseconds)

Performs a micro wait for the specified interval.

These waits occur in an ordinary busy loop, and as such are easily influenced by the runtime environment, such as execution in RAM, execution in ROM, memory cache on or off, etc. These wait times are therefore not very accurate.

These waits are not the same as an OS WAIT state. The system state remains as RUN state.

5.6 Interprocessor Management Functions

Interprocessor management functions are MP T-Kernel functions that realize synchronization between processors and exclusive control.

They include the spin lock control and atomic function, and a memory barrier function. These are low level operation functions realized by functions provided by hardware (refer to "2.10 Low Level Operation Functions"). Therefore, although the following standard specification is provided, details are implementation-defined, and there are also functions for which implementation is not feasible. If functions not in the standard specifications are added, however, the function names must be different from those given here.

Interprocessor management functions are provided by the library function or C language macros. These can be called from a task-independent portion and while dispatching and interrupts are disabled.

[Additional Notes]

It is assumed interprocessor management functions will be used with system software. Normal applications do not specifically conduct synchronization and the communication between processors in SMP T-Kernel.

5.6.1 Spinlock Control

Spinlocks are functions for exclusively controlling shared resources between processors.

The spinlock variable is locked prior to the use of shared resources. If the spinlock variable is already locked, it waits in busy wait until the lock is released.

[Additional Notes]

Spinlocks wait is busy wait and cannot be executed during the time processing such as other tasks. It must be noted that a decrease in processing efficiency results if busy waits frequently occur. Moreover, the lock period should be as short as possible.

(1) Spinlock Variables

In the spinlock, a T_SPLOCK spinlock variable and a T_RWSPLOCK read write spinlock variable are defined. The T_SPLOCK is used for spinlock operations and the T_RWSPLOCK is used for read write spinlock operations. These details are implementation-defined.

In SMP T-Kernel, it is preferable that the spinlock variable and the read write spinlock variable be able to allocate without distinguishing other variables in normal memory. However, this is not applied when there are hardware restrictions and details are implementation-defined.

The spinlock variable and the read write spinlock variable are initialized prior to use.

ER InitSpinLock(T_SPLOCK *lock)

Initializes the spinlock variable lock.

When the lock cannot be used for the spinlock due to hardware restrictions, etc., an error is returned. However, the detection of errors is implementation-defined.

E_MACV Specified memory area cannot be used for the spinlock

ER InitRWSpinLock(T_RWSPLOCK *lock)

Initializes the read write spinlock variable lock.

When the lock cannot be used for the read write spinlock due to hardware restrictions, etc., an error is returned. However, the detection of errors is implementation-defined.

E_MACV Specified memory area cannot be used for the read write spinlock

(2) Spinlock Operations

ER SpinLock(T_SPLOCK *lock)

Locks the spinlock variable lock. If it is already locked it waits in busy wait until the lock is released.

ER SpinTryLock(T_SPLOCK *lock)

Locks the spinlock variable lock. If it is already locked, it returns an error.

ER SpinUnlock(T_SPLOCK *lock)

Releases the lock for the spinlock variable lock.

E_BUSY	Already locked
E_NOSPT	Unsupported function (Spinlock is not supported)

(3) Spinlock Operations with Disable Interrupts

ER ISpinLock(T_SPLOCK *lock, UINT *intsts)

Locks the spinlock variable lock after external interrupts are prohibited. The state prior to the prohibition of interrupts is stored in intsts. If it is already locked, it waits in busy wait until the lock is released. The interrupt is prohibited while the lock is waiting for acquisition.

ER ISpinTryLock(T_SPLOCK *lock, UINT *intsts)

Locks the spinlock variable lock after external interrupts are prohibited. The state prior to the prohibition of interrupts is stored in intsts. If it is already locked, it returns to the state prior to the prohibition of the interrupt and returns an error.

ER ISpinUnlock(T_SPLOCK *lock, UINT intsts)

Releases the lock on the spinlock variable lock to permit external interrupts. Interrupts are permitted by returning to the state stored in intsts. intsts is the value returned by ISpinLock() or ISpinTryLock().Therefore, if the state is disable interrupt prior to locking, the interrupt is not permitted even if ISpinUnlock() is executed.

E_BUSY	Already locked
E_NOSPT	Unsupported function (Spinlock is not supported)

[Additional Notes]

The period when the spinlock variable is locked must be shortened as much as possible from the standpoint of processing efficiency of the overall system. Therefore, it is necessary to avoid entering wait with the acquired lock and dispatching to other tasks, etc. In order to do that, there is a disable interrupt during the locked period as a method. Spinlocks involving disable interrupt locks after disable interrupt and dispatches of tasks while locked is controlled.

ISpinLock() and ISPinUnlock() are equal to the following processing (However, the following codes do not consider errors).

```
ER ISpinLock(T_SPLOCK *lock, UINT *intsts )
{
      UW imask;

      DI ( imask );
       *intsts = imask;
      return SpinLock( lock );
}

ER ISpinUnlock( T_SPLOCK *lock, UINT intsts )
{
      ER    rtn;

      rtn = SpinUnlock ( lock );
      EI ( intsts );
      return rtn;
}
```

(4) Read Write Spinlock Operations

The read write spinlock only carries out exclusive control when there are multiple write locks (writing lock). Read lock (reading lock) permits multiple locks.

ER ReadLock(T_RWSPLOCK *lock)

Read locks the read write spinlock variable lock. The read lock can be locked multiple times. If it is already write locked, it waits in busy wait until the write lock is released.

ER ReadTryLock(T_RWSPLOCK *lock)

Read locks the read write spinlock variable lock. The read lock can be locked multiple times. If it is already locked, an error is returned.

ER ReadUnlock(T_RWSPLOCK *lock)

Releases the read lock for the read write spinlock variable lock.

ER WriteLock(T_RWSPLOCK *lock)

Write locks the read write spinlock variable lock. The write lock cannot be locked multiple times. If it is already read locked or write locked, it waits in busy wait until all the locks are released.

ER WriteTryLock(T_RWSPLOCK *lock)

Write locks the read write spinlock variable lock. If it is already read locked or write locked, an error is returned.

ER WriteUnlock(T_RWSPLOCK *lock)

Releases the write lock for the read write spinlock variable lock.

(5) Read Write Spinlock Operations With Disable Interrupts

ER IReadLock(T_RWSPLOCK *lock, UINT *intsts)

Read locks the read write spinlock variable lock after external interrupts are prohibited. The state prior to the prohibition of the interrupt is stored in intsts. The read lock can be locked multiple times. If it is already write locked, it waits in busy wait until the write lock is released. Interrupts are prohibited while the lock is waiting for acquisition.

ER IReadTryLock(T_RWSPLOCK *lock, UINT *intsts)

Read locks the read write spinlock variable lock after external interrupts are prohibited. The state prior to the prohibition of the interrupt is stored in intsts. The read lock can be locked multiple times. If it is already write locked, it returns to the state prior to the prohibition of the interrupt, and an error is returned.

ER IReadUnlock(T_RWSPLOCK *lock, UINT *intsts)

Releases the read lock on the read write spinlock variable lock to permit external interrupts. Interrupts are permitted by returning to the state stored in intsts. intsts is the value returned by IReadLock() or IReadTryLock().Therefore, if the state is disable interrupt prior to locking, the interrupt is not permitted even if IReadUnlock() is executed.

ER IWriteLock(T_RWSPLOCK *lock, UINT *intsts)

Write locks the read write spinlock variable lock after external interrupts are prohibited. The state prior to the prohibition of interrupts is stored in intsts. Write lock cannot be locked multiple times. If it is already read locked or write locked, it waits in busy wait until all locks are released. Interrupts are prohibited while the lock is waiting for acquisition.

ER IWriteTryLock(T_RWSPLOCK *lock, UINT *intsts)

Write locks the read write spinlock variable lock after external interrupts are prohibited. The state prior to the prohibition of interrupts is stored in intsts. Write lock cannot be locked multiple times. If it is already read locked or write locked, it returns to the state prior to the prohibition of interrupts, and an error is returned.

ER IWriteUnlock(T_RWSPLOCK *lock, UINT *intsts)

Releases the write lock on the read write spinlock variable lock to permit external interrupts. Interrupts are permitted by returning to the state stored in intsts. intsts is the value returned by IWriteLock() or IWriteTryLock().Therefore, if the state is disable interrupt prior to locking, the interrupt is not permitted even if IWriteUnlock() is executed.

5.6.2 Atomic Function

The atomic function is a memory operation function that guarantees inseparability in multiprocessors. Although the implementation of the function depends on hardware, the functions for the following memory barriers must be satisfied.

- (On source code) The execution results for memory operations positioned before the atomic function are monitored by all processors before any memory operations are executed by the atomic function.
- (On source code) The execution results for memory operations by the atomic function are monitored by all processors before code located after the atomic function is executed.

UW atomic_inc(UW *addr)

*addr++ The *addr after execution is the return value.

UW atomic_dec(UW *addr)

*addr-- The *addr after execution is the return value.

UW atomic_add(UW *addr, UW val)

*addr += val The *addr after execution is the return value.

UW atomic_sub(UW *addr, UW val)

*addr -= val The *addr after execution is the return value.

UW atomic_xchg(UW *addr, UW val)

*addr = vsl The *addr before execution is the return value.

UW atomic_cmpxchg(UW *addr, UW val, UW cmp)

If *addr == cmp, then *addr = val. The *addr before execution is the return value.

UW atomic_bitset(UW *addr, UW setptn)

*addr |= setptn The *addr before execution is the return value.

UW atomic_bitclr(UW *addr, UW clrptn)

*addr &= clrptn The *addr before execution is the return value.

[Additional Notes]

The atomic function guarantees that operations on memory shared between processors are done inseparably. At the same time, the atomic function serves as a memory barrier.

5.6.3 Memory Barriers

Depending on the processor, there may be cases where the order in the program code does not match the order in which memory is actually accessed such as cases in which instructions are permuted by out of order execution, etc. In operations on memory shared between processors, etc., there is the possibility of unforeseen operations. The order of the memory access can be guaranteed by memory barrier functions.

void mp_memory_barrir(void)

Prior to execution of the code after the memory barrier, it is guaranteed that the execution result of memory operations prior to memory barriers will be monitored from all processors.

[Additional Notes]

In processors which do not require memory barriers (order of the memory access is always guaranteed), the function of the memory barrier is ignored from the standpoint of program portability and an error will not be returned.

5.7 Power Management Functions

Functions called from the T-Kernel/OS to realize power-saving functions (refer to tk_set_pow()).

The manner of calling these functions is implementation-dependent. Simple system calls are possible, as is the use of a trap. Use of an extended SVC or other means that makes use of OS functions is not possible, however. Providing these functions in T-Monitor is another option.

The specifications given here for low-pow and off-pow are reference specifications. Since these functions are used only inside T-Kernel, other specifications may be devised as well. It is even possible to design completely different specifications in order to realize more advanced power-saving features. If the functionality is similar to that specified here, however, it would be best to follow these specifications as closely as practical.

This processor independent function is called in T-Kernel, and there are no specification differences between SMP T-Kernel and the T-Kernel 1.00 Specification.

void low_pow (void)

Switches to low-power mode and waits for an interrupt to be raised.

This function is called from the task dispatcher, and performs the following processing.

1. Goes to low-power mode.
2. Waits for an external interrupt to be raised.
3. When an external interrupt is raised, restores normal power mode and returns to its caller.

This function is called in interrupts disabled state. Interrupts must not be enabled. The speed of response to an interrupt affects processing speed, and should be as fast as possible.

void off_pow (void)

Suspends the system. When a resume factor occurs, it resumes system operation.

This function is called from tk_set_pow, and performs the following processing.

1. Puts the hardware in suspended state.
2. Waits for a resume factor to occur.
3. When a resume factor occurs, returns from suspended state and returns to its caller.

This function is called in interrupts disabled state. Interrupts must not be enabled. The device drivers perform the suspending and resuming of peripherals and other devices.

5.8 System Configuration Information Management Functions

System configuration information management functions are provided for storing, managing and making available information about the system configuration (maximum number of tasks, etc.) and any other information. These are not functions for adding or modifying information when the system is running.

How the system configuration information is to be stored is not specified here, but it is generally put in memory (ROM/RAM). This functionality is therefore not intended for storing large amounts of information.

Standard definitions are specified for some system configuration information, but additional information may be defined and used for applications, subsystems, or device drivers.

The format of system configuration information consists of a name and defined data as a set.

- Name
  The name is a string of up to 16 characters.
  Characters that can be used (UB) are a to z, A to Z, 0 to 9 and (under bar).
- Defined data
  Data consists of numbers (integers) or character strings.
  Characters that can be used (UB) are any characters other than 0x00 to 0x1F, 0x7F, or 0xFF (character codes).

Sample:	Name	Defined Data
	SysVer	1 0
	SysName	T-Kernel Version 1.00

5.8.1 System Configuration Information Acquisition

System configuration information is acquired by using extended SVC. This function is used inside T-Kernel, and can also be used by applications, subsystems, device drivers and so on. Use inside T-Kernel does not have to go through extended SVC; this choice is implementation-dependent.

```
INT tk_get_cfn ( UB *name, INT *val, INT max )
```

Name	Name
val	Array storing numeric strings
max	Number of elements in val array
return cord	Defined numeric information count or error

Gets numeric string information from system configuration information. This function gets up to max items of numerical string information defined by the name specified in the name parameter and stores the acquired information in val. The number of items of defined numeric string information is passed in the return code. If return code > max, this indicates that not all the information could be stored. By specifying max = 0, the number of numeric strings can be found out without actually storing them in val.

E_NOEXS is returned if no information is defined with the name specified in the name parameter. The behavior if the information defined as name is a character string is indeterminate.

This function can be invoked from any protection level, without being limited by the protection level from which T-Kernel/OS system call can be invoked.

```
INT tk_get_cfs ( UB *name, UB *buf, INT max )
```

Name	Name
buf	Array storing character string
max	Maximum size of buf (in bytes)
return code	Size of defined character string information (in bytes) or error

Gets character string information from system configuration information. This function gets up to max characters of character string information defined by the name specified in the name parameter and stores the acquired information in buf. If the acquired character string is shorter than max characters, it is terminated by '¥0' when stored. The length of the defined character string information (not including '¥0') is passed in the return code. If return code > max, this indicates that not all the information could be stored. By specifying max = 0, the character string length can be found out without actually storing anything in buf.

E_NOEXS is returned if no information is defined with the name specified in the name parameter. The behavior if the information defined as name is a numeric string is indeterminate.

This function can be invoked from any protection level, without being limited by the protection level from which T-Kernel/OS system call can be invoked.

5.8.2 Standard System Configuration Information

The following information is defined as standard system configuration information. A standard information name is prefixed by T.

N: Numeric string information
S: Character string information

・Product information

S: TSysName System name (product name)

・Processor Information

N: TSysPrNum	Processor count for the entire system
N: TKerPrNum	Processor count controlled by the kernel

・Maximum object counts

N: TMaxTskId	Maximum tasks
N: TMaxSemId	Maximum semaphores
N: TMaxFlgId	Maximum event flags
N: TMaxMbxId	Maximum mailboxes
N: TMaxMtxId	Maximum mutexes
N: TMaxMbfId	Maximum message buffers
N: TMaxPorId	Maximum rendezvous ports
N: TMaxMpfId	Maximum fixed-size memory pools
N: TMaxMplId	Maximum variable-size memory pools
N: TMaxDomId	Maximum domain count
N: TMaxCycId	Maximum cyclic handlers
N: TMaxAlmId	Maximum alarm handlers
N: TMaxResId	Maximum resource groups
N: TMaxSsyId	Maximum subsystems
N: TMaxSsyPri	Maximum subsystem priorities

・Other

N: TsysStkSz	Default system stack size (in bytes)
N: TSVCLimit	Lowest protection level for system call invoking
N: TOAPLimit	The lowest protection level in which access protection is invalid
N: TTimPeriod	Timer interrupt interval (ms)

・Device management

N: TMaxRegDev	Maximum device registrations
N: TMaxOpnDev	Maximum device open count
N: TMaxReqDev	Maximum device requests
N: TDEvtMbfSz	Event notification message buffer size (in bytes) Maximum event notification message length (in bytes) If TDEvtMbfSz is not defined or if the message buffer size is a negative value, an event notification message buffer is not used.

When multiple values are defined for any of the above numeric strings, they are stored in the same order as in the explanation.

Examples: tk_get_cfn("TDEvtMbfSz", val, 2)
val[0] = Event notification message buffer size
val[1] = Maximum event notification message length

[Items Concerning SMP T-Kernel]

The following system configuration information was added in SMP T-Kernel.

N: TSysPrNum	Overall system processor count
N: TKerPrNum	Processor count controlled by the kernel
N: TMaxDomId	Maximum domain count
N: TOAPLimit	Lowest effective valid protection level

TSysPrNum and TKerPrNum have the same value in situations in which SMP T-Kernel only is operating. In the future where systems with multiple MP T-Kernel are combined together, TKerPrNum will indicate the processor count controlled by a specific SMP T-Kernel in the system and TSysPrNum will indicate the processor count in the entire system.

Chapter 6 Starting SMP T-Kernel

6.1 Subsystem and Device Driver Starting

There are no differences in the specification for starting subsystems and device drivers with the T-Kernel 1.00 Specification.
Entry routines like the following are defined for subsystems and device drivers.

```
ER main( INT ac, UB *av[] )
{
        if ( ac >= 0 ) {
                /* Subsystem/device driver start processing */
        } else {
                /* Subsystem/device driver termination processing */
        }
        return ercd;
}
```

This entry routine simply performs startup processing or termination processing for a subsystem or device driver and does not provide any actual service. It must return to its caller as soon as the startup processing or termination processing is performed. An entry routine must perform its processing as quickly as possible and return to its caller.

An entry routine is called by the task which belongs to the system resource group at the time of normal system startup or shutdown, and runs in the context of the OS start processing task or termination processing task (protection level 0). In some OS implementations, it may run as a quasi-task portion. In a system that supports dynamic loading of subsystems and device drivers, it may be called at other times besides system startup and shutdown.

When there are multiple subsystems and device drivers, each of the entry routine is called one at a time at system startup and shutdown. In no case are multiple entry routines called by different tasks at the same time. Accordingly, if subsystem or device driver initialization needs to be performed in a certain order, this order can be maintained by completing all necessary processing before returning from an entry routine.

The entry routine function name is normally main, but any other name may be used if, for example, main cannot be used because of linking with the OS.

The methods of registering entry routines with the OS, specifying parameters, and specifying the order in which entry routines are called are all dependent on the OS implementation.

・Startup processing

ac	Number of parameters (≧0)
av	Parameters (string)
return code	Error

A value of ac>=0 indicates startup processing. After performing the subsystem or device driver initialization, it registers the subsystem or device driver.

Passing of a negative value (error) as the return code means the startup processing failed. Depending on the OS implementation, the subsystem or device driver may be deleted from memory, so error must not be returned while registering subsystem or device driver. The registration must first be removed before returning the error. Allocated resources must also be released. They are not released automatically.

The parameters ac and av are the same as the parameters passed to the standard C language main() function, with ac indicating the number of parameters and av indicating a parameter string as an array of ac + 1 pointers. The array termination (av[ac]) is NULL..

av[0] is the name of the subsystem or device driver. Generally this is the file name of the subsystem or device driver, but the kind of name in which it is stored is implementation-dependent. It is also possible to have no name (blank string "").

Parameters from av[1] onwards are defined separately for each subsystem and device driver.

After exiting from the entry routine, the character string space specified by av is deleted, so parameters must be saved to a different location if needed.

・Termination processing

ac	-1
av	NULL
return code	Error

A value of ac < 0 indicates termination processing. After deleting the subsystem or device driver registration, the entry routine releases allocated resources. If an error occurs during termination processing, the processing must not be aborted but must be completed to the extent possible. If some of the processing could not be completed normally, an error is passed in the return code.

The behavior if termination processing is called while requests to the subsystem or device driver are being processed is dependent on the subsystem or device driver implementation. Generally, termination processing is called at system shutdown and requests are not issued during processing. For this reason, ordinary behavior is not guaranteed in the case of requests issued during termination processing.

Chapter 7 SMP T-Kernel/DS Functions

This chapter gives detailed explanations of the functions provided by SMP T-Kernel Debugger Support (Debugger Support).
SMP T-Kernel/DS provides functions enabling a debugger to reference SMP T-Kernel internal states and run a trace. The functions provided by SMP T-Kernel/DS are only for debugger use and not for use by applications or other programs.

General cautions and notes

- Except where otherwise noted, SMP T-kernel/DS service calls (td_) can be called from a task independent portion and while dispatching and interrupts are disabled.
  There may be some limitations, however, imposed by specific implementations.

- When SMP T-Kernel/DS service calls (td_) are invoked in interrupts disabled state, they are processed without enabling interrupts. Other OS states likewise remain unchanged during this processing.
  When it is called in the enable external interrupts state and the enable dispatch state, the OS state may change because the OS operation is also continued.

- SMP T-Kernel/DS service calls (td_) cannot be invoked from a lower protection level than that at which T-Kernel/OS system calls can be invoked (lower than TSVCLimit)(E OACV).

- The access protection function of the object is not applied to SMP T-Kernel/DS service calls (td_). Therefore, they can be executed for all objects.

- Error codes such as E_PAR, E_MACV, and E_CTX that always have the possibility of occurring are not described here unless there is some special reason for doing so.

- Detection of error codes E_PAR, E_MACV, and E_CTX is implementation-dependent; these may not always be detected as error. For this reason, the service calls must not be invoked in such a way that these errors might occur.

7.1 Kernel Internal State Reference Functions

These functions enable a debugger to get T-Kernel internal states. They include functions for getting a list of objects, getting task precedence, getting the order in which tasks are queued, getting the status of objects, system and task registers, and getting time.

td_lst_tsk, td_lst_sem, td_lst_flg, td_lst_mbx, td_lst_mtx, td_lst_mbf, td_lst_por, td_lst_mpf, td_lst_mpl, td_lst_cyc, td_lst_alm, td_lst_dom, td_lst_ssy

Reference Object ID List

[C Language Interface]

```
INT ct = td_lst_tsk ( ID list[], INT nent ) ;      /* task */
INT ct = td_lst_sem ( ID list[], INT nent ) ;      /* semaphore */
INT ct = td_lst_flg ( ID list[], INT nent ) ;      /* event flag */
INT ct = td_lst_mbx ( ID list[], INT nent ) ;      /* mailbox */
INT ct = td_lst_mtx ( ID list[], INT nent ) ;      /* mutex */
INT ct = td_lst_mbf ( ID list[], INT nent ) ;      /* message buffer */
INT ct = td_lst_por ( ID list[], INT nent ) ;      /* rendezvous port */
INT ct = td_lst_mpf ( ID list[], INT nent ) ;      /* fixed-size memory pool */
INT ct = td_lst_mpl ( ID list[], INT nent ) ;      /* variable-size memory pool */
INT ct = td_lst_cyc ( ID list[], INT nent ) ;      /* cyclic handler */
INT ct = td_lst_alm ( ID list[], INT nent ) ;      /* alarm handler */
INT ct = td_lst_dom ( ID list[], INT nent ) ;      /* Domain */
INT ct = td_lst_ssy ( ID list[], INT nent ) ;      /* subsystem */
```

[Parameters]

ID	list[]	Location of object ID list
INT	nent	Maximum number of list entries to retrieve

[Return Parameters]

INT	ct	Number of objects used orError Code

[Description]

Gets a list of IDs of objects currently being used, and puts up to nent IDs into list. The number of objects used is passed in the return code. If return code > nent, this means that not all IDs could be retrieved in this system call.

[Items Concerning SMP T-Kernel]

td_lst_dom for acquiring the list of domain ID's was newly added.

There are no differences between other calls and the T-Kernel 1.00 Specification.

td_rdy_que

Get Task Precedence

[C Language Interface]

INT ct = td_rdy_que (PRI pri, ID list[], INT nent) ;

[Parameters]

PRI	pri	Task priority
ID	list[]	Location of task ID list
INT	nent	Maximum number of list entries

[Return Parameters]

INT	ct	Number of priority pri tasks in a run state orError Code

[Description]

Gets a list of IDs of the tasks in a run state (READY state or RUN state) whose task priority is pri, arranged in order from highest to lowest precedence.

This function stores in the location designated in list up to nent task IDs, arranged in order of precedence starting from the highest-precedence task ID at the head of the list.

The number of tasks in a run state with priority pri is passed in the return code. If return code > nent, this means that not all task IDs could be retrieved in this call.

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

td_sem_que, td_flg_que, td_mbx_que, td_mtx_que, td_smbf_que, td_rmbf_que, td_cal_que, td_acp_que, td_mpf_que, td_mpl_que

Reference Queue

[C Language Interface]

```
INT ct = td_sem_que ( ID semid, ID list[], INT nent ) ;    /* semaphore */
INT ct = td_flg_que ( ID flgid, ID list[], INT nent ) ;    /* event flag */
INT ct = td_mbx_que ( ID mbxid, ID list[], INT nent ) ;    /* mailbox */
INT ct = td_mtx_que ( ID mtxid, ID list[], INT nent ) ;    /* mutex */
INT ct = td_smbf_que ( ID mbfid, ID list[], INT nent ) ;   /* message buffer send */
INT ct = td_rmbf_que ( ID mbfid, ID list[], INT nent ) ;   /* message buffer receive */
INT ct = td_cal_que ( ID porid, ID list[], INT nent ) ;    /* rendezvous call */
INT ct = td_acp_que ( ID porid, ID list[], INT nent ) ;    /* rendezvous accept */
INT ct = td_mpf_que ( ID mpfid, ID list[], INT nent ) ;    /* fixed-size memory pool */
INT ct = td_mpl_que ( ID mplid, ID list[], INT nent ) ;    /* variable-size memory pool */
```

[Parameters]

ID	～ID	Object ID
ID	list[]	Location of waiting task IDs
INT	nent	Maximum number of list entries

[Return Parameters]

INT	ct	Number of waiting tasks
	or	Error Code

[Error Codes]

E_ID	Bad identifier
E_NOEXS	Object does not exist

[Description]

Gets a list of IDs of tasks waiting for the object designated in --id. This function stores in the location designated in list up to nent task IDs, arranged in the order in which tasks are queued, starting from the first task in the queue. The number of queued tasks is passed in the return code. If return code > nent, this means that not all task IDs could be retrieved in this system call.

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

Reference Task Status td_ref_tsk

[C Language Interface]

```
ER ercd = td_ref_tsk ( ID tskid, TD_RTSK *rtsk );
```

[Parameters]

ID	tskid	Task ID (TSK_SELF can be specified)
TD_RTSK	rtsk	Address of Packet for returning the task state

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Bad identifier
E_NOEXS	Object does not exist

[Description]

Gets the state of the task designated in tskid. This function is similar to tk_ref_tsk, with the task start address and stack information added to the state information obtained.

```
typedef struct td_rtsk {
        VP        exinf;      /* extended information */
        PRI       tskpri;     /* current priority */
        PRI       tskbpri;    /* base priority */
        UINT      tskstat;    /* task state */
        UINT      tskwait;    /* wait factor */
        ID        wid;        /*   waiting object ID */
        INT       wupcnt;     /* queued wakeup request count */
        INT       suscnt;     /* SUSPEND request nesting count */
        RELTIM    slicetime;  /* maximum continuous run time (ms) */
        UINT      waitmask;   /* masked wait factors */
        UINT      texmask;    /* allowed task exceptions */
        UINT      tskevent;   /* raised task event */
        FP        task;       /* task start address */
        INT       stksz;      /* user stack size (in bytes) */
        INT       sstksz;     /* system stack size (in bytes) */
        VP        istack;     /* user stack pointer initial value */
        VP        isstack;    /* system stack pointer initial value */
} TD_RTSK;
```

The stack area extends from the stack pointer initial value toward the low addresses for the number of bytes designated as the stack size.

```
istack  - stksz   <=  user stack area    <  istack
isstack - sstksz  <=  system stack area  <  isstack
```

Note that the stack pointer initial value (istack, isstack) is not the same as its current position. The stack area may be used even before a task is started. Calling td_get_reg gets the correct value of stack pointer.

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

td_ref_sem , td_ref_flg, td_ref_mbx, td_ref_mtx, td_ref_mbf, td_ref_por, td_ref_mpf, td_ref_mpl, td_ref_cyc, td_ref_alm, td_ref_ssy

Reference Queue

[C Language Interface]

```
ER ercd = td_ref_sem ( ID semid, TD_RSEM *rsem );      /* semaphore */
ER ercd = td_ref_flg ( ID flgid, TD_RFLG *rflg );      /* event flag */
ER ercd = td_ref_mbx ( ID mbxid, TD_RMBX *rmbx );      /* mailbox */
ER ercd = td_ref_mtx ( ID mtxid, TD_RMTX *rmtx );      /* mutex */
ER ercd = td_ref_mbf ( ID mbfid, TD_RMBF *rmbf );      /* message buffer */
ER ercd = td_ref_por ( ID porid, TD_RPOR *rpor );      /* rendezvous port */
ER ercd = td_ref_mpf ( ID mpfid, TD_RMPF *rmpf );      /* fixed-size memory */
ER ercd = td_ref_mpl ( ID mplid, TD_RMPL *rmpl );      /* variable-size memory pool */
ER ercd = td_ref_cyc ( ID cycid, TD_RCYC *rcyc );      /* cyclic handler */
ER ercd = td_ref_alm ( ID almid, TD_RALM *ralm );      /* alarm handler */
ER ercd = td_ref_dom ( ID domid, TD_RDOM *rdom );      /* Domain */
ER ercd = td_ref_ssy ( ID   ssid, TD_RSSY *rssy );     /* subsystem */
```

[Parameters]

ID	～id	Object ID
TD_R～	r～	Address of status information packet

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Bad identifier
E_NOEXS	Object does not exist

[Description]

Gets the status of an object. This is similar to tk_ref_-.
The return packets are defined as follows.

```
/*
 * Semaphore status information td_ref_sem
 */
typedef struct td_rsem {
      VP    exinf;     /* extended information */
      ID    wtsk;      /* waiting task ID */
      INT semcnt;      /* current semaphore count */
} TD_RSEM;

/*
 * Event flag status information      td_ref_flg
 */
typedef struct td_rflg {
      VP    exinf;     /* extended information */
      ID    wtsk;      /* waiting task ID */
      UINTflgptn;      /* current event flag pattern */
} TD_RFLG;

/*
 * Mailbox  status information      td_ref_mbx
 */
typedef struct td_rmbx {
      VP    exinf;     /* extended information */
```

```
        ID    wtsk;      /* waiting task ID */
        T_MSG      *pk_msg;/* next message to be received */
    } TD_RMBX;

    /*
     * Mutex status information      td_ref_mtx
     */
    typedef struct td_rmtx {
        VP    exinf;        /* extended information */
        ID    htsk;         /* locking task ID */
        ID    wtsk;         /* ID of task waiting for lock */
    } TD_RMTX;

    /*
     * Message buffer status information      td_ref_mbf
     */
    typedef struct td_rmbf {
        VP    exinf;        /* extended information */
        ID    wtsk;         /* receive waiting task ID */
        ID    stsk;         /* send waiting task ID */
        INT msgsz;          /* size (in bytes) of next message to be received */
        INT frbufsz;        /* free buffer size (in bytes) */
        INT maxmsz;         /* maximum message length (in bytes) */
    } TD_RMBF;

    /*
     * Rendezvous port status information      td_ref_por
     */
    typedef struct td_rpor {
        VP    exinf;        /* extended information */
        ID    wtsk;         /* call waiting task ID */
        ID    atsk;         /* acceptance waiting task ID */
        INT maxcmsz;        /* call message maximum length (in bytes) */
        INT maxrmsz;        /* accept message maximum length (in bytes) */
    } TD_RPOR;

    /*
     * Fixed-size memory pool status information      td_ref_mpf
     */
    typedef struct td_rmpf {
        VP    exinf;        /* extended information */
        ID    wtsk;         /* waiting task ID */
        INT frbcnt;         /* free block count */
    } TD_RMPF;

    /*
     * Variable-size memory pool status information      td_ref_mpl
     */
    typedef struct td_rmpl {
        VP    exinf;        /* extended information */
        ID    wtsk;         /* waiting task ID */
        INT frsz;           /* total free space (in bytes) */
        INT maxsz;          /* maximum contiguous free space (in bytes) */
    } TD_RMPL;

    /*
     * Cyclic handler status information td_ref_cyc
     */
    typedef struct td_rcyc {
        VP    exinf;              /* extended information */
```

```
        RELTIM    lfttim;      /* time remaining until next handler start */
        UINTcycstat;           /* cyclic handler status */
    } TD_RCYC;

    /*
     * Alarm handler status information td_ref_alm
     */
    typedef struct td_ralm {
        VP    exinf;           /* extended information */
        RELTIM    lfttim;      /* time remaining until next handler start */
        UINTalmstat;           /* alarm handler status */
    } TD_RALM;

    /*
     * Domain status information td_ref_dom
     */
    typedef struct td_rdom {
        VP    exinf;        /* Extended information */
        ID    domid;        /* Domain ID to which the domain belongs */
        UINT    objcnt;   /* Object count belonging to the domain */
    } TD_RDOM;

    /*
     * Subsystem status information td_ref_ssy
     */
    typedef struct td_rssy {
        PRI ssypri;         /* subsystem priority */
        INT resblksz;       /* resource control block size (in bytes) */
    } TD_RSSY;
```

[Items Concerning SMP T-Kernel]

td_ref_dom that refers to the state of the domain was newly added.

There are no differences between other calls and the T-Kernel 1.00 Specification.

Reference Task Exception Status td_ref_tex

[C Language Interface]

```
ER ercd = td_ref_tex ( ID tskid, TD_RTEX *pk_rtex );
```

[Parameters]

ID	tskid	Task ID (TSK_SELF can be designated)
TD_RTEX*	pk_rtex	Packet address for returning the task exception status

[Return Parameters]

ERercd	Error code

pk_rtex detail:

UINT	pendtex	Raised task exceptions
UINT	texmask	Allowed task exceptions

[Error Codes]

E_OK	Normal completion
E_ID	Bad identifier
E_NOEXS	Object does not exist

[Description]

Gets the task exception status. This is similar to tk_ref_tex.

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

Reference Task Statistics td_inf_tsk

[C Language Interface]

ER ercd = td_inf_tsk (ID tskid, TD_ITSK *pk_itsk, BOOL clr);

[Parameters]

ID	tskid	Task ID (TSK_SELF can be designated)
TD_ITSK*	pk_itsk	Address of packet for returning task statistics
BOOL	clr	Task statistics clear flag

[Return Parameters]

ER	ercd	Error code

pk_itsk detail:

RELTIM	stime	Cumulative system-level run time (ms)
RELTIM	utime	Cumulative user-level run time (ms)

[Error Codes]

E_OK	Normal completion
E_ID	ID number is invalid
E_NOEXS	Object does not exist

[Description]

Gets task statistics. This is similar to tk_inf_tsk. When clr = TRUE (=? 0), accumulated information is reset (cleared to 0) after getting the statistics.

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

Get Task Register td_get_reg

[C Language Interface]

ER ercd = td_get_reg (ID tskid, T_REGS *pk_regs, T_EIT *pk_eit, T_CREGS *pk_cregs) ;

[Parameters]

ID	tskid	Task ID (TSK_SELF cannot be designated)

[Return Parameters]

T_REGS	pk_regs	General registers
T_EIT	pk_eit	Registers saved when exception is raised
T_CREGS	pk_cregs	Control registers
ER	ercd	Error code

The contents of T_REGS, T_EIT, and T_CREGS are defined for each CPU and implementation.

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_OBJ	Invalid object state (issued for current RUN state task)

[Description]

Gets the register values of the task designated in tskid. This is similar to tk_get_reg.

Registers cannot be referenced for the task currently in RUN state. Except when a task-independent portion is executing, the current RUN state task is the invoking task.

When NULL is designated for regs, eit, or cregs, the corresponding register is not referenced.

The contents of T_REGS, T_EIT, and T_CREGS are implementation-dependent.

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

Set Task Register td_set_reg

[C Language Interface]

ER ercd = td_set_reg (ID tskid, T_REGS *pk_regs, T_EIT *pk_eit, T_CREGS *pk_cregs) ;

[Parameters]

ID	tskid	Task ID (TSK_SELF cannot be specified)
T_REGS	pk_regs	General registers
T_EIT	pk_eit	Registers saved when exception is raised
T_CREGS	pk_cregs	Control registers

The contents of T_REGS, T_EIT, and T_CREGS are defined for each CPU and implementation.

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task specified in tskid does not exist)
E_OBJ	Invalid object state (issued for current RUN state task)

[Description]

Sets registers of the task designated in tskid. This is similar to tk_set_reg.

Registers cannot be set for the task currently in RUN state. Except when a task-independent portion is executing, the current RUN state task is the invoking task.

When NULL is designated for regs, eit, or cregs, the corresponding register is not set.

The contents of T_REGS, T_EIT, and T_CREGS are implementation-dependent.

[Items Concerning SMP T-Kernel]

There are no differences from the T-Kernel 1.00 Specification.

Reference System Status td_ref_sys

[C Language Interface]

```
ER ercd = td_ref_sys ( TD_RSYS *pk_rsys ) ;
```

[Parameters]

TD_RSYS* pk_rsys Packet to Refer System Address of packet for returning status information

[Return Parameters]

ER ercd Error code

pk_rsys detail:

INT	sysstat	System status
ID	runtskid	ID of current RUN state task
ID	schedtskid	ID of task scheduled to go to RUN state

[Error Codes]

E_OK Normal completion

[Description]

Gets the system status. This is similar to tk_ref_sys.

[Items Concerning SMP T-Kernel]

In SMP T-Kernel, a RUN state exists for each respective processor. Therefore, information on the processor that issued this call is returned. Information on other processors cannot be acquired.

Get System Time td_get_tim

[C Language Interface]
ER ercd = td_get_tim (SYSTIM *tim, UNIT *ofs) ;

[Parameters]
SYSTIM* tim Address of packet for returning current time (ms)
UNIT* ofs Location for returning elapsed time from tim (nanoseconds)

[Return Parameters]
ER ercd Error code

tim_detail: Current time (ms)
ofs_detail: Elapsed time from tim (nanoseconds)

[Error Codes]
E_OK Normal completion

[Description]
Gets the current time as total elapsed milliseconds since 0:00:00 (GMT), January 1, 1985. The value returned in tim is the same as that obtained by tk_get_tim. tim is expressed in the resolution of timer interrupt intervals (cycles), but even more precise time information is obtained in ofs as the time elapsed from tim in nanoseconds. The resolution of ofs is implementation-dependent, but is generally the timer hardware resolution.

Since tim is time counted based on timer interrupts, in some cases time is not refreshed, when a timer interrupt cycle arrives while interrupts are disabled and the timer interrupt handler is not started (is delayed). In such cases, the time as updated by the previous timer interrupt is returned in tim, and the time elapsed from the previous timer interrupt is returned in ofs. Accordingly, in some cases, ofs will be a larger value than the timer interrupt cycle. The length of elapsed time that can be measured by ofs depends on the hardware, but it should preferably be able to measure at least up to twice the timer interrupt cycle (0 <= ofs < twice the timer interrupt cycle).

Note that the time returned in tim and ofs is the time at some point between the calling of and return from td_get_tim. It is neither the time at which td_get_tim was called nor the time of return from td_get_tim. In order to obtain more accurate information, this function should be called in interrupts disabled state.

[Items Concerning SMP T-Kernel]
There are no differences from the T-Kernel 1.00 Specification.

Get System Operating Time td_get_otm

[C Language Interface]
ER ercd = td_get_otm (SYSTIM *tim, UINT *ofs) ;

[Parameters]
SYSTIM* tim Address of packet for returning operating time (ms)
UNIT* ofs Location for returning elapsed time from tim (nanoseconds)

[Return Parameters]
ER ercd Error code
tim_detail
Operating time (ms)
ofs_detail:
Elapsed time from tim (nanoseconds)

[Error Codes]
E_OK Normal completion

[Description]
Gets the system operating time (uptime, as elapsed milliseconds since the system was booted). The value returned in tim is the same as that obtained by tk_get_otm. tim is expressed in the resolution of timer interrupt intervals (cycles), but even more precise time information is obtained in ofs as the time elapsed from tim in nanoseconds. The resolution of ofs is implementation-dependent, but is generally the timer hardware resolution.

Since tim is time counted based on timer interrupts, in some cases time is not refreshed, when a timer interrupt cycle arrives while interrupts are disabled and the timer interrupt handler is not started (is delayed). In such cases, the time as updated by the previous timer interrupt is returned in tim, and the time elapsed from the previous timer interrupt is returned in ofs. Accordingly, in some cases, ofs will be a larger value than the timer interrupt cycle. The length of elapsed time that can be measured by ofs depends on the hardware, but it should preferably be able to measure at least up to twice the timer interrupt cycle (0 <= ofs < twice the timer interrupt cycle).

Note that the time returned in tim and ofs is the time at some point between the calling of and return from td_get_otm. It is neither the time at which td_get_otm was called nor the time of return from td_get_otm. In order to obtain more accurate information, this function should be called in interrupts disabled state.

[Items Concerning SMP T-Kernel]
There are no differences from the T-Kernel 1.00 Specification.

Refer to DS Object Name td_ref_dsname

[C Language Interface]

ER ercd = td_ref_dsname(UINT type, ID id, UB *dsname);

[Parameters]

UINT type object type
ID id object ID
UB *dsname address to return DS Object name

[Return Parameters]

ER ercd Error code
dsname detail: DS Object name, set at object creation or by td_set_dsname()

[Error Codes]

E_OK	Normal completion
E_PAR	Invalid object type
E_NOEXS	Object does not exist
E_OBJ	DS Object name is not used

[Description]

Get DS Object name (dsname), which is set at object creation. The object is specified by object type (type) and object ID (id).)

Object types (type) are as follows:

```
TN_TSK 0x01 /* task */
TN_SEM 0x02 /* semaphore */
TN_FLG 0x03 /* event flag */
TN_MBX 0x04 /* mail box */
TN_MBF 0x05 /* message buffer */
TN_POR 0x06 /* rendezvous port */
TN_MTX 0x07 /* mutex */
TN_MPL 0x08 /* variable-size memory pool */
TN_MPF 0x09 /* fixed-size memory pool */
TN_CYC 0x0a /* cyclic handler */
TN_ALM 0x0b /* alarm handler */
TN_DOM 0x0c /* Domain */
```

[Items Concerning SMP T-Kernel]

In the T-Kernel 1.00 Specification, the DS Object name is used for debugging but in SMP T-Kernel this was abolished and the object name is used. Although the DS Object name is permitted for use only in debugging, and overlapping of the name is permitted, the object name is used in functions such as retrieving objects and overlapping of the name is not permitted in the same type of object within the same domain.

In this call, the object name is used for the above change instead of the DS Object name. It can be said that there is no difference for this call which acquires the name.

Set DS Object Name td_set_dsname

[C Language Interface]

ER ercd = td_set_dsname(UINT type, ID id, UB *dsname);

[Parameters]

UINT	type	object type
ID	id	object ID
UB	*dsname	DS Object name to be set

[Return Parameters]

ERercd Error code

[Error Codes]

E_OK	Normal completion
E_PAR	Invalid object type
E_NOEXS	Object does not exist
E_OBJ	DS Object name is not used
E_ONAME	Specified object name has already been used

[Description]

Update DS Object name (dsname), which is set at object creation. The object is specified by object type (type) and object ID (id).)

Object types (type) are as same as that of td_ref_dsname().

Object name is valid if TA OSNAME is set as the object attribute. When the object where TA_ONAME attribute is not specified is targeted, E_OBJ is returned.

The object name is a character string of 8 characters or less. 1 character is 1 byte long and when it is not equal to 8 bytes, 0's are used to fill the remaining bytes. The characters that can be used are a-z, A-Z, and 0-9 but a check of character codes is not done in SMP T-Kernel.

The object name does not allow repetition in the same type of object in the same domain. When there is an overlap, E_ONAME is returned.

[Items Concerning SMP T-Kernel]

In the T-Kernel 1.00 Specification, the DS Object name is used for debugging but in SMP T-Kernel this is abolished and the object name is used. Although the DS Object name is only permitted for use in debugging and the overlapping of names was permitted, the object name is used in the functions such as retrieving objects, etc. and the overlapping of names is not permitted in the same type of object in the same domain.

In this call, the object name is used for the above change instead of the DS Object name. Therefore, the point that the overlapping of names is checked is different from the T-Kernel 1.00 Specification.

Moreover, in the T-Kernel 1.00 Specification the DS debugging name was used only for debugging but the object name also is used in the application. It must be noted that there is an effect on the application due to the change of object names.

td_fnd_dom, td_fnd_tsk, td_fnd_sem,td_fnd_flg, td_fnd_mbx, td_fnd_mtx, td_fnd_mbf, td_fnd_por, td_fnd_mpf, td_fnd_mpl, td_fnd_alm, td_fnd_cyc

Retrieval of Each Object ID

td_fnd_xxx: Find ObjectID

[C Language Interface]

```
ID domid = td_fnd_dom ( ID domid, UB *oname ) ;    /* Domain */
ID tskid = td_fnd_tsk ( ID domid, UB *oname ) ;    /* Task */
ID semid = td_fnd_sem ( ID domid, UB *oname ) ;    /* Semaphore */
ID flgid = td_fnd_flg ( ID domid, UB *oname ) ;    /* Event flag */
ID mbxid = td_fnd_mbx ( ID domid, UB *oname ) ;    /* Mailbox */
ID mtxid = td_fnd_mtx ( ID domid, UB *oname ) ;    /* Mutex */
ID mbfid = td_fnd_mbf ( ID domid, UB *oname ) ;    /* Message buffer */
ID porid = td_fnd_por ( ID domid, UB *oname ) ;    /* Rendezvous port */
ID mpfid = td_fnd_mpf ( ID domid, UB *oname ) ;    /* Fixed size memory pool */
ID mplid = td_fnd_mpl ( ID domid, UB *oname ) ;    /* Variable-size memory pool */
ID almid = td_fnd_alm ( ID domid, UB *oname ) ;    /* Alarm handler */
ID cycid = td_fnd_cyc ( ID domid, UB *oname ) ;    /* Cyclic handler */
```

[Parameters]

ID	domid	Domain ID
UB*	oname	Object name

[Return Parameters]

ID	～id	Specified object ID
or	error code	

[Error Codes]

E_ID	Invalid ID number (domid is invalid or cannot be used)
E_NOEXS	Object does not exist (object of oname does not exist)
E_PAR	Parameter error (oname is invalid or cannot be used)

[Description]

Retrieves the object belonging to the domain shown by domid by using the object name, and acquires the object ID. It is the same as tk_fnd_xxx().

In oname, the object name of the object to be retrieved is specified.

When the object specified by domid and oname is discovered, the ID of that object is returned. When the corresponding object does not exist, E_NOEXS is returned.

Only accessible objects can be retrieved. The retrieval of objects that cannot be accessed due to access protection does not succeed, and E_NOEXS is returned.

[Items Concerning SMP T-Kernel]

This call does not exist in the T-Kernel 1.00 Specification.

td_dmi_dom, td_dmi_tsk, td_dmi_sem,td_dmi_flg, td_dmi_mbx, td_dmi_mtx, td_dmi_mbf, td_dmi_por, td_dmi_mpf, td_dmi_mpl, td_dmi_alm, td_dmi_cyc
Get Domain Information for Each Object

td_dmi_xxx: Get Domain Information

[C Language Interface]

```
ER ercd = td_dmi_dom ( ID domid, TD_DMI *pk_dmi ) ;    /* Domain */
ER ercd = td_dmi_tsk ( ID tskid, TD_DMI *pk_dmi ) ;    /* Task */
ER ercd = td_dmi_sem ( ID semid, TD_DMI *pk_dmi ) ;    /* Semaphore */
ER ercd = td_dmi_flg ( ID flgid, TD_DMI *pk_dmi ) ;    /* Event flag */
ER ercd = td_dmi_mbx ( ID mbxid, TD_DMI *pk_dmi ) ;    /* Mailbox */
ER ercd = td_dmi_mtx ( ID mtxid, TD_DMI *pk_dmi ) ;    /* Mutex */
ER ercd = td_dmi_mbf ( ID mbfid, TD_DMI *pk_dmi ) ;    /* Message buffer */
ER ercd = td_dmi_por ( ID porid, TD_DMI *pk_dmi ) ;    /* Rendezvous port */
ER ercd = td_dmi_mpf ( ID mpfid, TD_DMI *pk_dmi ) ;    /* Fixed size memory pool */
ER ercd = td_dmi_mpl ( ID mplid, TD_DMI *pk_dmi ) ;    /* Variable-size memory pool */
ER ercd = td_dmi_alm ( ID almid, TD_DMI *pk_dmi ) ;    /* Alarm handler */
ER ercd = td_dmi_cyc ( ID cycid, TD_DMI *pk_dmi ) ;    /* Cyclic handler */
```

[Parameters]

ID	～id	ObjectID	Domain ID
TD_DMI*	pk_dmi	Packet to Domain Information	Packet address where domain information is returned

[Return Parameters]

ER ercd	ErrorCode	Error code

pk_dmi details

ATR	domatr	DomainAttribute	Domain attribute
ID	domid	DomainID	ID of the domain to which it belongs
ID	kdmid	Kernel Domain ID	Kernel domain ID to which it belongs
UB	oname[8]	Object name	Object name

[Error Codes]

E_ID	Invalid ID number (~id is invalid or cannot be used)
E_NOEXS	Object does not exist (object of ~id does not exist)
E_PAR	Parameter error (Value for which the packet address for the return parameter cannot be used)
E_DACV	ACCESS PROTECTION VIOLATION

[Description]

Gets information related to the domain of the target object displayed by ~id. It is the same as tk_dmi_xxx().

The attribute related to the domain of the target object is set in domatr. domatr takes the following values.

domatr := [TA_ONAME]|(TA_PRIVATE || TA_PROTECTED || TA_PUBLIC)

TA_ONAME	Object name is specified
TA_PROTECTED	Access protection attribute is protect
TA_PRIVATE	Access protection attribute is private
TA_PUBLIC	Access protection attribute is public

The ID number of the domain to which the target object belongs is set in domid.

The ID number of the kernel domain to which the target object belongs is set in kdmid.

The object name of the target object is set in oname. When the object name is not set to the target object, all contents are 0.

Access protection is applied to this system call.

[Items Concerning SMP T-Kernel]

This call does not exist in the T-Kernel 1.00 Specification.

7.2 Trace Functions

These functions enable a debugger to trace program execution. Execution trace is performed by setting hook routines.

・Return from a hook routine must be done after states have returned to where they were when the hook routine was called. Restoring of registers, however, can be done in accordance with the register saving rules of C language functions.

・In a hook routine, limitations on states must not be modified to make them less restrictive than when the routine was called. For example, if the hook routine was called during interrupts disabled state, interrupts must not be enabled.

・A hook routine was called at protection level 0.

・A hook routine inherits the stack at the time of the hook. Too much stack use may therefore cause a stack overflow. The extent to which the stack can be used is not definite, since it differs with the situation at the time of the hook. Switching to a separate stack in the hook routine would be safer.

Define System Call/Extended SVC Hook Routine td_hoc_svc

[C Language Interface]
ER ercd = td_hok_svc (TD_HSVC *hsvc) ;

[Parameters]
TD_HSVC hsvc Hook routine definition information

hsvc detail:
FP enter Hook routine before calling the service call
FP leave Hook routine after calling the service call

[Return Parameters]
ER ercd Error code

[Description]

Sets hook routines before and after the issuing of a system call or extended SVC. Setting NULL in hsvc cancels a hook routine.

The objects of a trace are T-Kernel/OS system calls (tk_-) and extended SVC. Depending on the implementation, generally tk_ret_int is not the object of a trace.

T-Kernel/DS service calls (td_-) are not objects of a trace.

A hook routine is called in the context from which the system call or extended SVC was called. For example, the invoking task in a hook routine is the same as the task that invoked the system call or extended SVC.

Since task dispatching and interrupts can occur inside system call processing, enter() and leave() are not necessarily called in succession as a pair in every case. If a system call is one that does not return, leave() will not be called.

VP enter(FN fncd, TD_CALINF *calinf, ...)

fncd Function code
< 0 System call
>= 0 Extended SVC
calinf Caller information
... Parameters (variable number)
return code Any value to be passed to leave()

typedef struct td_calinf {
As information for determining the address from which a system call or extended SVC was called, it is preferable to include information for performing a stack back-trace. The contents are implementation-dependent but generally consist of register values such as stack pointer and program counter.
} TD_CALINF;

This is called right before a system call or extended SVC.

The value passed in the return code is passed on to the corresponding leave(). This makes it possible to confirm the pairing of enter() and leave() calls or to pass any other information.

exinf = enter(fncd, &calinf, ...)
ret = system call or extended SVC execution
leave(fncd , ret, exinf)

・System call:

The parameters are the same as the system call parameters.

Example: For system call tk_wai_sem(ID semid, INT cnt, TMO tmout)
enter(TFN_WAI_SEM, &calinf, semid, cnt, tmout)

・Extended SVC:

The parameters are as in the packet passed to the extended SVC handler.
fncd is the same as that passed to the extended SVC handler.

enter(FN fncd, TD_CALINF *calinf, VP pk_para)

void leave(FN fncd, INT ret, VP exinf)

fncd Function code
ret Return code of the system call or extended SVC
exinf Any value returned by enter()

This is called right after returning from a system call or extended SVC.

When a hook routine is set after a system call or extended SVC is called (while the system call or extended SVC is executing), in some cases, only leave()may be called without calling enter(). In such a case, NULL is passed in exinf.
If, on the other hand, a hook routine is canceled after a system call or extended SVC is called, there may be cases when enter() is called but not leave().

[Items Concerning SMP T-Kernel]
There are no differences from the T-Kernel 1.00 Specification.

Define Task Dispatch Hook Routine td_hoc_dsp

[C Language Interface]
ER ercd = td_hok_dsp (TD_HDSP *hdsp) ;

[Parameters]
TD_HDSP hdsp Hook routine definition information

hdsp_detail:
FP exec Hook routine when execution starts
FP stop Hook routine when execution stops

[Return Parameters]
ER ercd Error code

[Description]
Sets hook routines in the task dispatcher. A hook routine is canceled by setting NULL in hdsp.

The hook routines are called in dispatch disabled state. The hook routines must not invoke T-Kernel/OS system calls (tk_-) or extended SVC. T-Kernel/DS service calls (td_-) may be invoked.

void exec(ID tskid, INT lsid)

tskid Task ID of the started or resumed task
lsid Logical ID of the task designated in tskid

This is called when the designated task starts execution or resumes. At the time exec() is called, the task designated in tskid is already in RUN state and logical space has been switched. However, execution of the tskid task program code occurs after the return from exec().

void stop(ID tskid, INT lsid, UINT tskstat)

tskid Task ID of the task stopping execution
lsid Logical ID of the task designated in tskid
tskstat State of the task designated in tskid

This is called when the designated task stops execution. tskstat indicates the task state after stopping as one of the following states.

TTS_RDY READY state
TTS_WAI WAIT state
TTS_SUS SUSPEND state
TTS_WAS WAIT-SUSPEND state
TTS_DMT DORMANT state
0 NON-EXISTENT state

At the time stop() is called, the task designated in tskid has already entered the state indicated in tskstat. The logical space is indeterminate.

[Items Concerning SMP T-Kernel]
There are no differences from the T-Kernel 1.00 Specification.

Define Interrupt handler Hook Routine td_hoc_int

[C Language Interface]
ER ercd = td_hok_int (TD_HINT *hint) ;

[Parameters]
TD_HINT hint Hook routine definition information

hint_detail:
FP enter Hook routine before calling the handler
FP leave Hook routine after calling the handler

[Return Parameters]
ER ercd Error code

[Description]
Sets hook routines before and after an interrupt handler is called. Hook routine setting cannot be done independently for different exception or interrupt factors. One pair of hook routines is set in common for all exception and interrupt factors.

Setting hint to NULL cancels the hook routines.

The hook routines are called as task-independent portion (part of the interrupt handler). Accordingly, the hook routines can call only those system calls that can be invoked from a task-independent portion.

Note that hook routines can be set only for interrupt handlers defined by tk_def_int with the TA_HLNG attribute. A TA_ASM attribute interrupt handler cannot be hooked by a hook routine. Hooking of a TA_ASM attribute interrupt handler is possible only by directly manipulating the exception/interrupt vector table. The actual methods are implementation-dependent.

void enter(UINT dintno)
void leave(UINT dintno)

dintno Interrupt definition number

The parameters passed to enter() and leave() are the same as those passed to the exception/interrupt handler. Depending on the implementation, information other than dintno may also be passed.

A hook routine is called as follows from a high-level language support routine.

```
enter(dintno);
inthdr(dintno); /* exception/interrupt handler */
leave(dintno);
```

enter() is called in interrupts disabled state, and interrupts must not be enabled. Since leave() assumes the status on return from inthdr(), the interrupts disabled or enabled status is indeterminate.
enter() can obtain only the same information as that obtainable by inthdr(). Information that cannot be obtained by inthdr() cannot be obtained by enter(). The information that can be obtained by enter() and inthdr() is guaranteed by the specification to include dintno, but other information is implementation-dependent. Note that since interrupts disabled state and other states may change while leave() is running, leave() does not necessarily obtain the same information as that obtained by enter() or inthdr().

[Items Concerning SMP T-Kernel]
There are no differences from the T-Kernel 1.00 Specification.

Chapter 8 Reference

8.1 List of Error Codes

——— Normal Completion Error Class (0) ———

E_OK 0 Normal completion

——— Internal Error Class (5 to 8) ———

E_SYS ERCD(-5, 0) System error

An error of unknown cause affecting the system as a whole.

E_NOCOP ERCD(-6, 0) The specified co-processor cannot be used

This error code is returned when the specified co-processor is not installed in the currently running hardware, or abnormal co-processor operation was detected.

——— Unsupported Error Class (9 to 16) ———

E_NOSPT ERCD(-9, 0) Unsupported function

When some system call functions are not supported and such a function was specified, error code E_RSATR or E_NOSPTS is returned. If E_RSATR does not apply, error code E_NOSPT is returned.

E_RSFN ERCD(-10, 0) Reserved function code number

This error code is returned when it is attempted to execute a system call specifying a reserved function code (undefined function code), and also when it is attempted to execute an undefined extended SVC handler.

E_RSATR ERCD(-11, 0) Reserved attribute

This error code is returned when an undefined or unsupported object attribute is specified.

Checking for this error may be omitted if system-dependent optimization is implemented.

——— Parameter Error Class (17 to 24) ———

E_PAR ERCD(-17, 0) Parameter error

Checking for this error may be omitted if system-dependent optimization is implemented.

E_ID ERCD（-18, 0） Invalid ID number

E_ID is an error that occurs only for objects having an ID number.

Error code E_PAR is returned when a static error is detected in the parameter, such as reserved number or out of range for parameters such as interrupt definition numbers.

——— Call Context Error Class (25 to 32) ———

E_CTX ERCD(-25, 0) Context error

This error indicates that the specified system call cannot be issued in the current context (task portion/task-independent portion or handler RUN state).

This error must be issued whenever there is a meaningful context error in issuing a system call, such as calling from a task-independent portion a system call that may put the invoking task in WAIT state. Due to implementation limitations, there may be other system calls that when called from a given context (such as an interrupt handler) will cause this error to be returned.

E_MACV ERCD(-26, 0) Memory cannot be accessed; memory access privilege error

Error detection is implementation-dependent.

E_OACV ERCD(-27, 0) Object access privilege error

This error code is returned when a user task tries to manipulate a system object.

The definition of system objects and error detection are implementation-dependent.

E_ILUSE ERCD(-28, 0) System call illegal use

—— Resource Constraint Error Class (33 to 40) ——

E_NOMEM ERCD(-33, 0) Insufficient memory

This error code is returned when there is insufficient memory (no memory) for allocating an object control block space, user stack space, memory pool space, message buffer space, etc.

E_LIMIT ERCD(-34, 0) System limit exceeded

This error code is returned when it is attempted to create more of an object than the system allows.

—— Object State Error Class (41 to 48) ——

E_OBJ ERCD(-41, 0) Invalid object state

E_NOEXS ERCD(-42, 0) Object does not exist

E_QOVR ERCD(-43, 0) Queuing or nesting overflow

—— Wait Error Class (49 to 56) ——

E_RLWAI ERCD(-49, 0) WAIT state released

E_TMOUT ERCD(-50, 0) Polling failed or timeout

E_DLT ERCD(-51, 0) The object being waited for was deleted

E_DISWAI ERCD(-52, 0) Wait released by wait disabled state

—— Device Error Class (57 to 64) (T-Kernel/SM) ——

E_IO ERCD(-57, 0) IO error

- Error information specific to individual devices may be defined in E_IO sub-codes.

E_NOMDA ERCD(-58, 0) No media

—— Status Error Class (65 to 72) (T-Kernel/SM) ——

E_BUSY ERCD(-65, 0) Busy

E_ABORT ERCD(-66, 0) Processing was aborted

E_RONLY ERCD(-67, 0) Write protected

—— Domain Error Class (68～70) (MP T-Kernel) ——

E_DOMAIN ERCD(-68, 0) Domain error

This error indicates that an operation is not permitted due to a difference in the domain when it attempted to operate an object that belongs to another domain.

In AMP T-Kernel, this error is returned when a system call that cannot be used between processors is issued for an object that belongs to the domain of other processors and when the domains of other processors are specified for the domain to which it belongs in system calls for object creation. However, when ID of the specified domain is invalid and the domain does not exist, E_ID error or E_NOEXS is returned instead of this error.

E_ONAME ERCD(-69, 0) Object name error

This error indicates that the specified object name has already been used in the domain.

E_DACV ERCD(-70, 0) Access protection error

This error indicates that the operation is not permitted due to access protection when it attempted to operate an object that belongs to another domain.

——— Error Class Between Processors (71～73) (MP T-Kernel) ———

E_IPC ERCD(-71, 0) Interprocessor communication error

This error indicates that a failure occurred in some sort of communication between processors during the execution of a system call, and the execution result of the system call was not understood.

This error results from the answer of another processor not being properly received. When this error is returned, the result of the system call is not guaranteed. There is also the possibility that the requested operation is executing in another processor. If the failure of the requested operation can be distinguished, an E_IPCA error or E_IPCS error which is described below is returned instead of this error.

E_IPCA ERCD(-72, 0) Absolute interprocessor communication error

This error indicates that a failure occurred in some sort of communication between processors during the execution of a system call and the requested system call ended unsuccessfully. The difference between the E_IPC error is this error guarantees the execution result of the system call is a failure.

E_IPCS ERCD(-73, 0) Interprocessor communication status error

This error indicates that communication between processors is not possible due to some reason.
This error is returned when communication between processors cannot be done in normal state such as the other processor is stopped or is initializing. When communication between processors is not possible in a failed state, E_IPCA is returned.