T-Kernel 2.0 Specification

December 2014

T-Engine Forum

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

T-Kernel Specification Version 2.01.00

For information about this specification, please contact the following:

T-Engine Forum Secretariat
In YRP Ubiquitous Networking Laboratory
28th Kowa Building, 2-20-1 Nishi-gotanda
Shinagawa, Tokyo
Japan 141-0031
TEL: +81-(0)-3-5437-0572
FAX: +81-(0)-3-5437-2399
E-mail: office@t-engine.org

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
2.00.00	2011-04-01	Initial release.	T-Engine Forum
2.00.01	2012-11-21	• Fixed specification of ChkSpaceBstrR and CheckSpaceBstrRW to return the length of the accessible string in bytes, not in TRON code characters. • Replaced the terms 'access privilege information' and 'access privilege' with 'caller access privilege information', to clarify their meanings to avoid misunderstandings. • Corrected SetTaskSpace description to be more accurate, and clarified its ambiguous portions. • Fixed a few typographical errors.	T-Engine Forum
2.00.02	2013-02-01	• Removed descriptions on function for canceling wakeup requests, which is non-existent. • Moved some descriptions on tk_rel_wai from footnote to the main body of the document, in order to clarify that the behavior is defined as a part of this specification.	T-Engine Forum
2.00.03	2013-12-18	• Fixed a typo: ac[0] -> av[0]	T-Engine Forum

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
2.01.00	2014-11-20	• Clarification and clean up of the use of the following names: T-Kernel/OS, T-Kernel/SM, T-Kernel/DS • Cleaning up explanations of error codes that can be generated in implementation-dependent manner. • BOOL is now typedef of unsigned int. • Additional explanation is provided for relative time and system time. • E_NOMEM is removed from the list of the errors returned by tk_dly_tsk. • Spurious explanation is removed from the explanation of tk_clr_flg. • CONST modifier is added to the argument msg of tk_fwd_por and tk_rpl_rdv. • Display position of the note to tk_rot_rdq is changed. • CONST modifier is added to the argument addr of ChkSpaceR, ChkSpaceRW, and ChkSpaceRE. • Explanation is enhanced for the return error codes for ChkSpaceBstrR, ChkSpaceBstrRW, ChkSpaceTstrR, and ChkSpaceTstrR. • Explanations for parameters are modified: – wtsk: Wait Task Information -> Waiting Task ID – stsk: Send Task Information -> Send Waiting Task ID – atsk: Accept Task Information -> Accept Waiting Task ID – nblk: Number of Block -> Number of Blocks – nmemb: Number of Memory Block -> Number of Memory Blocks – nreq: Number of Request -> Number of Requests	T-Engine Forum

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
2.01.00	2014-11-20	• Fixed typos: – MAKER -> maker – unable to write to attr -> unable to write to addr – blockcount -> blockcont – pk_calm, almatr, or almhdr is invalid -> pk_calm or almhdr is invalid – 5.11.1.1 Parameter -> Parameter – 5.11.2.1 Parameter -> Parameter	T-Engine Forum

Contents

List of Figures

List of Tables

API Notation

In the parts of this specification that describe APIs, the specification of each API (Application Program Interface) is explained in the format illustrated below. In addition to system calls that directly call kernel functions, APIs include functions implemented as extended SVCs (extended system calls), macros, and libraries.

API Name - Description

This is an API name and its description.

C Language Interface

This is an API's C language interface and header file(s) to include.

Parameter

Describes an API's parameter(s), i.e. information passed to the T-Kernel when the API is issued.

Return Parameter

Describes an API's return parameter(s), i.e. information returned by the T-Kernel when the execution of the API ends.

A return parameter that is returned as an API's function value may be called "return code." A return parameter can include, besides return code, a value stored at a pointer that points at memory location where some information can be stored.

Error Code

Describes errors that can occur in an API.

The following error codes are common to all APIs and are not included in the error code listings for each API:

E_SYS , E_NOSPT , E_RSFN , E_MACV , E_OACV.

The detection of the error conditions that may result in the following error codes is implementation-dependent; such conditions may not always be detected as errors:

E_PAR , E_MACV , E_CTX.

Error code E_CTX is included in the error code section of individual API only when API can encounter an error due to a semantically wrong caller context: e.g., the case of task-independent portion's calling an API that can block. If an API's constraints in the caller's context are implementation-dependent, and such semantic errors are not universal across all implementations, the explanation of E_CTX is not included in the error section of the API under discussion.

Implementations may generate errors that are not explained in the explanation section of error codes.

Valid Context

Indicates the context (task portion, quasi-task portion, and task-independent portion) that can issue the API under consideration.

Description

Describes the API 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).

[x]
: x is optional.

Example of Using Parameters Notation

```
wfmode := (TWF_ANDW || TWF_ORW) | [TWF_CLR]
```

The above description means that 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 approach and specification.

Difference from T-Kernel 1.0

Explains the difference(s) between T-Kernel 1.0.

Index of T-Kernel/OS System Calls

The T-Kernel/OS system calls described in this specification are listed below in alphabetical order.

- tk_get_rid - Get Task Resource ID
- tk_get_tid - Get Task Identifier
- tk_get_tim - Get Time
- tk_get_tim_u - Get Time (in Microseconds)
- tk_get_tsp - Get Task Space
- tk_inf_tsk - Reference Task Statistics
- tk_inf_tsk_u - Reference Task Statistics (Microseconds)
- tk_loc_mtx - Lock Mutex
- tk_loc_mtx_u - Lock Mutex (in Microseconds)
- tk_ras_tex - Raise Task Exception
- tk_rcv_mbf - Receive Message from Message Buffer
- tk_rcv_mbf_u - Receive Message from Message Buffer (in Microseconds)
- tk_rcv_mbx - Receive Message from Mailbox
- tk_rcv_mbx_u - Receive Message from Mailbox (in Microseconds)
- tk_ref_alm - Reference Alarm Handler Status
- tk_ref_alm_u - Reference Alarm Handler Status (Microseconds)
- tk_ref_cyc - Reference Cyclic Handler Status
- tk_ref_cyc_u - Reference Cyclic Handler Status (Microseconds)
- tk_ref_flg - Reference Event Flag Status
- tk_ref_mbf - Reference Message Buffer Status
- tk_ref_mbx - Reference Mailbox Status
- tk_ref_mpf - Reference Fixed-size Memory Pool Status
- tk_ref_mpl - Reference Variable-size Memory Pool Status
- tk_ref_mtx - Refer Mutex Status
- tk_ref_por - Reference Port Status
- tk_ref_sem - Reference Semaphore Status
- tk_ref_ssy - Reference Subsystem Status
- tk_ref_sys - Reference System Status
- tk_ref_tex - Reference Task Exception Status
- tk_ref_tsk - Reference Task Status
- tk_ref_tsk_u - Reference Task Status (Microseconds)
- tk_ref_ver - Reference Version Information
- tk_rel_mpf - Release Fixed-size Memory Block
- tk_rel_mpl - Release Variable-size Memory Block
- tk_rel_wai - Release Wait

Index of T-Kernel/SM Extended SVC and Libraries

The T-Kernel/SM extended SVC and libraries described in this specification are listed below in alphabetical order.

- eventfn - Event function
- execfn - Execute function
- GetPhysicalTimerConfig - Get Physical Timer Configuration Information
- GetPhysicalTimerCount - Get Physical Timer Count
- GetSpaceInfo - Get Various Information about Address Space
- in_b - Read from I/O Port (in Bytes)
- in_d - Read from I/O Port (in Double-words)
- in_h - Read from I/O Port (in Half-words)
- in_w - Read from I/O Port (in Words)
- isDI - Get Interrupt Disable Status
- Kcalloc - Allocate Resident Memory
- Kfree - Release Resident Memory
- Kmalloc - Allocate Resident Memory
- Krealloc - Reallocate Resident Memory
- Lock - Lock Fast Lock
- LockSpace - Lock Memory Space
- low_pow - Move System to Low-power Mode
- MapMemory -Map Memory
- MLock - Lock Fast Multi-lock
- MLockTmo - Lock Fast Multi-lock (with Timeout)
- MLockTmo_u - Lock Fast Multi-lock (with Timeout, in Microseconds)
- MUnlock - Unlock Fast Multi-lock
- off_pow - Move System to Suspend State
- openfn - Open function
- out_b - Write to I/O Port (in Bytes)
- out_d - Write to I/O Port (in Double-words)
- out_h - Write to I/O Port (in Half-words)
- out_w - Write to I/O Port (in Words)
- SetCacheMode - Set Cache Mode
- SetIntMode - Set Interrupt Mode
- SetMemoryAccess - Set Memory Access Privilege
- SetOBJNAME - Set Object Name
- SetTaskSpace - Set Task Space
- StartPhysicalTimer - Start Physical Timer
- StopPhysicalTimer - Stop Physical Timer

Index of T-Kernel/DS System Calls

The T-Kernel/DS system calls described in this specification are listed below in alphabetical order.

Chapter 1

T-Kernel Overview

1.1 Position of T-Kernel

The position of T-Kernel in the overall T-Engine system is shown in Figure 1.1, “Positioning for T-Kernel”.

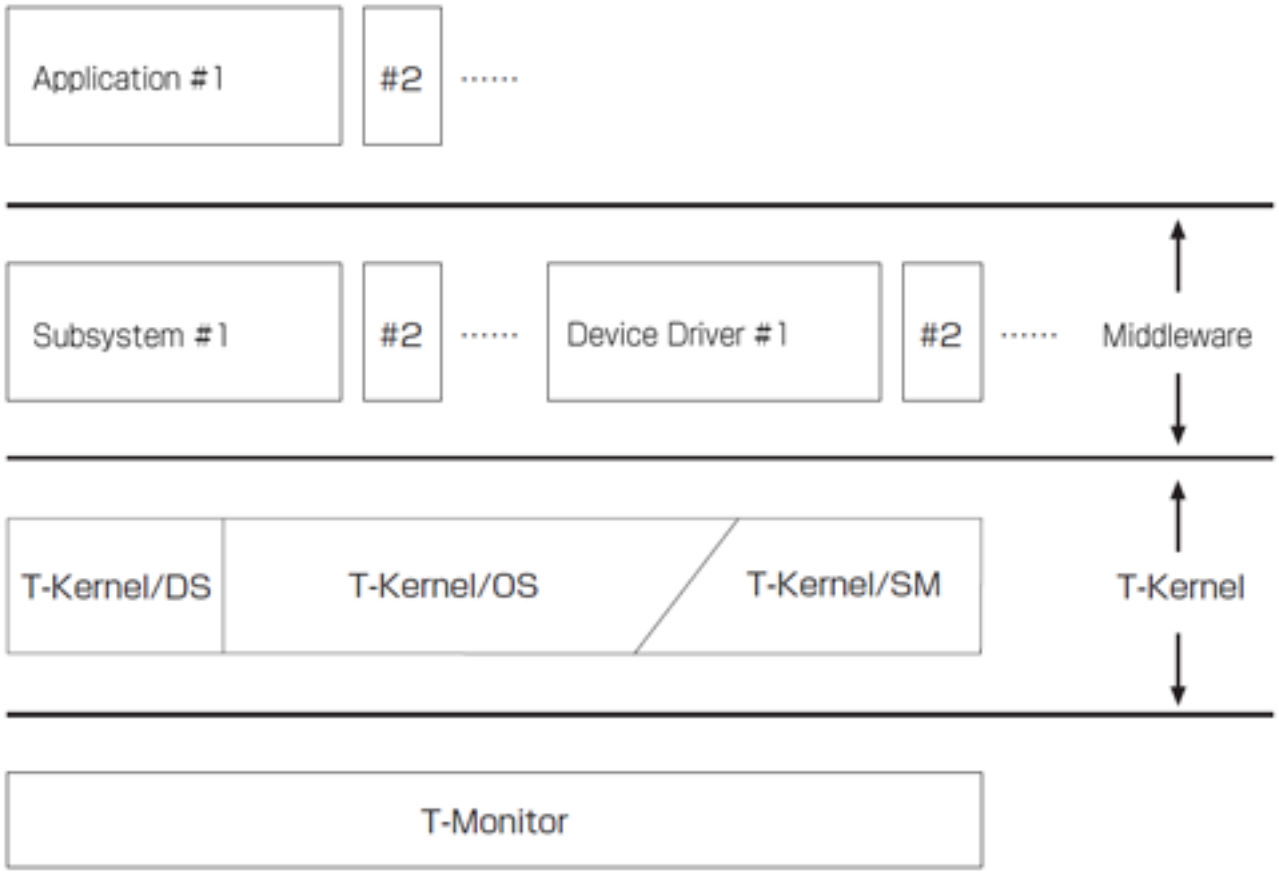

Figure 1.1: Positioning for T-Kernel

T-Kernel generally refers to all of T-Kernel/OS (Operating System), T-Kernel/SM (System Manager), and T-Kernel/DS); but in some cases T-Kernel/OS only (narrow definition) is called T-Kernel.

T-Kernel/OS provides the following functions:

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

- Subsystem Management Functions

T-Kernel/SM provides the following kinds of functions:

- System Memory Management Functions
- Address Space Management Functions
- Device Management Functions
- Interrupt Management Functions
- I/O Port Access Support Functions
- Power Management Functions
- System Configuration Information Management Functions
- Memory Cache Control Functions
- Physical Timer Functions
- Utility Functions

T-Kernel/DS provides the following kinds of functions exclusively for debugging use:

- Kernel Internal State Acquisition Functions
- Trace Functions

Difference from T-Kernel 1.0
Memory cache control functions, physical timer functions, and utility functions are functions that were added in T-Kernel 2.0.

1.2 Scalability

T-Kernel is a real-time operating system for embedded system use, applicable to a wide range of systems large and small. It is aimed at enhancing portability of software such as device drivers and middleware.

The T-Kernel specification is designed to be applicable even to large-scale systems. So there are features unnecessary for small systems. The approach of defining subsets has the disadvantage of hampering portability of device drivers, middleware and other software. Functional requirements also vary widely from one target system to another, making it difficult to settle on workable subset specifications.

The T-Kernel specification does not adopt a layer division or other subset approach. In principle, all the T-Kernel implementations must implement the specification in their entirety. However, the simple dummy implementation can be applied to those functions that are not required in the target systems.

A "Simple dummy implementation" means one that does not provide the entire range of specified functions but does not behave abnormally (return error, etc.) if a non-implemented function is called. It is important to provide an environment on which a middleware developed for a large system can run without modification. For example, a system that does not use an MMU (Memory Management Unit) can implement the T-Kernel/SM LockSpace() as follows:

```
#define LockSpace(addr, len)    ( E_OK )
```

The absence of an MMU does not, however, permits the implementor not to implement LockSpace() or to return the error code E_NOSPT.

At the same time, when middleware is designed or developed, leaving out LockSpace() from an implementation because the target system does not use an MMU would prevent the middleware from supporting a system that does use an MMU.

Providing users with the means for removing or skipping unnecessary functions is also allowed. However, the resulting T-Kernel is judged as modified T-Kernel.

Middleware vendors must note the following points:

- Middleware must be designed to meet all the T-Kernel requirements. It is important, in other words, that middleware developed for large-scale systems can run without problem on other systems.
- Providing users with the means for removing or skipping unnecessary functions is allowed.

1.3 T-Kernel 2.0 Overview

1.3.1 Positioning and Basic Policy of T-Kernel 2.0

As the T-Kernel (T-Kernel 1.0), which was published when T-Engine Forum was established, is showing proved achievement and steady increase of products adopting it, there are requests for additional functions to take advantage of the hardware with higher performance and more functionalities. To meet these requests, T-Kernel 2.0 specification for the real-time operating system was developed as a step to the new and wider deployment.

T-Kernel 2.0 specification is compatible with T-Kernel 1.0 specification to take advantage of the past T-Kernel achievements and forsters smooth migration of users to the enhanced kernel. T-Kernel 2.0 specification is not only source compatible but also binary compatible with T-Kernel 1.0 specification. For example, after T-Kernel 1.0 is upgraded to T-Kernel 2.0, existing device drivers, middleware, applications, etc. that were running on T-Kernel 1.0 will run on T-Kernel 2.0 without recompilation.

Additionally, for T-Kernel 2.0, readability and searchability of this specification are significantly improved by using XML-based document source, besides improving the expression and explanation through the review of the complicated or immature description.

1.3.2 Additional Functions to T-Kernel 2.0

1. Time management functions in microseconds

   While T-Kernel 1.0 used milliseconds for the time management functions such as the cyclic handler and alarm handler, and the time-related function such as the timeout, T-Kernel 2.0 adds APIs that handle them in microseconds.

   Since data with 32-bit width can handle only very short period of time when expressed in microseconds, data with 64-bit width is introduced as time-related parameters (see Introduction of 64-bit integer data type).

   For a name of API with 64-bit parameter in microseconds, "_u" is appended to the end of the corresponding API name in T-Kernel 1.0. u means μ. For 64-bit parameter in microseconds, "_u" is appended to the end of the parameter name also.

Example 1.1 Example of API for 64-bit microseconds

```
/* API of T-Kernel 1.0 for 32-bit milliseconds */
tk_sta_alm( ID almid, RELTIM almtim )

/* API of T-Kernel 2.0 for 64-bit microseconds */
tk_sta_alm_u( ID almid, RELTIM_U almtim_u )
```

   However, time handling APIs are not unified to use only microseconds in T-Kernel 2.0. According to the basic policy of keeping upward compatibility, APIs for milliseconds of T-Kernel 1.0 can still be used in T-Kernel 2.0 making both time units co-exist.

   Actual time resolution in T-Kernel time management functions uses one that is specified by the "timer interrupt interval" (`TTimPeriod`) in Section 5.7.2, “Standard System Configuration Information”. Therefore, the "timer interrupt interval" (`TTimPeriod`) must be set to an enough short period of time to precisely specify parameters for the time management functions in microseconds. For more details, see Section 5.7.2, “Standard System Configuration Information”.

   For comparison, while the maximum time length handled by 32-bit signed integer is approximately 24 days in milliseconds, it is approximately 35 minutes in microseconds. When using data with 64-bit width, virtually unlimited time length can be handled.

2. Support for large mass-storage device

   Some parameters of device management functions also can handle data with 64-bit width to support large mass-storage device such as a hard disk.

   For a name of API with 64-bit parameter, "_d" is appended to the end of the corresponding API name in T-Kernel 1.0. "_d" means double integer. For 64-bit parameter, "_d" is appended to the end of the parameter name also.

Example 1.2 Example of API with 64-bit Parameters

```
/* API of T-Kernel 1.0 */
tk_swri_dev( ID dd, W start, VP buf, W size, W *asize )

/* API of T-Kernel 2.0 with 64-bit Parameters */
tk_swri_dev_d( ID dd, D start_d, void *buf, W size, W *asize )
```

   For example, the maximum storage size that can be handled by T-Kernel 1.0 with 32-bit data width was approximately 1 TB (= 512-byte x MATH: 2⊠31) for a general hard disk with 512-byte block size. In T-Kernel 2.0, this limit is increased by the addition of API with 64-bit data width.

3. Introduction of 64-bit integer data type

   64-bit integer data type is introduced for API parameters to realize the features in preceding two sections. For this reason, T-Kernel specification adopts a long long data type that is formally specified as a part of C language standard.

   The name of data type that represents 64-bit integer is D for signed integer and UD for unsigned integer. 'D' means Double integer.

4. Other additional functions

   Cache-related function, physical timer function, utility function, etc. are added.

Chapter 2

T-Kernel Concepts

2.1 Meaning of Basic Terminology

Task, invoking task

The basic logical unit of concurrent program execution is called a "task." Whereas the code in one task is executed in sequence, codes in different tasks can be executed in parallel. This concurrent processing is a conceptual phenomenon, from the standpoint of applications; in actual implementation it is accomplished by time-sharing among tasks as controlled by the kernel.

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

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).

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.

Context

The environment in which a program runs is generally called "context." For a context to be called identical, at the very least the processor operation mode must be the same and the stack space must be the same (part of the same contiguous area). Note that context is a conceptual entity 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.

Precedence

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

Additional Notes
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 used in the specification to make clear the order in which processing is to be executed.
Precedence among tasks is determined based on task priority.

API and system call

The standard interfaces for calling functions provided by T-Kernel from applications or middleware are collectively called API (Application Program Interface). In addition to system calls that directly call kernel functions, APIs include functions implemented as extended SVCs, macros, and libraries.

An API that calls T-Kernel/OS or T-Kernel/DS is a system call while an API that calls T-Kernel/SM is extended SVC, macro, or library.

Kernel

Kernel refers to a combination of T-Kernel/OS and T-Kernel/DS by narrow definition. It refers to entire T-Kernel by wide definition.

T-Kernel/SM is not a kernel in a strict meaning because it is an extended function of T-Kernel/OS that uses subsystem functions of T-Kernel/OS.

T-Kernel or T-Kernel itself refers to a combination of T-Kernel/OS, T-Kernel/SM, and T-Kernel/DS.

Implementation-defined

That something is implementation-defined means that something is not standardized in the T-Kernel specification and should be defined for each implementation. The specifics of the implementation should be described clearly in the implementation specifications. In application programs, the portability for the portion dependent on implementation-defined items is not assured.

Implementation-dependent

That something is implementation-dependent means that in the T-Kernel specification, the behavior of something varies according to the target systems or system operating conditions. The behavior should be defined for each implementation. The specifics of the implementation should be described clearly in the implementation specifications. In application programs, the portion dependent on implementation-dependent items needs to be modified when porting in principle.

2.2 Task States and Scheduling Rules

2.2.1 Task States

Task states are classified primarily into the five below. Of these, Waiting 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 RUNNING state or READY state.

RUNNING 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 RUNNING state.

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 the highest precedence among the tasks in READY state.

Waiting states
: The task cannot run because the 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 Waiting states, the program counter and register values, and the 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 to the Waiting state. This state is subdivided into the following three states.

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

  SUSPENDED state
  : Execution was forcibly interrupted by another task.

  WAITING-SUSPENDED state
  : The task is in both WAITING state and SUSPENDED state at the same time. WAITING-SUSPENDED state results when another task requests suspension of a task already in WAITING state.

    T-Kernel makes a clear distinction between WAITING state and SUSPENDED state. A task cannot go to SUSPENDED state on its own.

DORMANT state
: The task has not yet been started or has completed execution. While a task is in DORMANT state, information presenting 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.

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 2.5, “System States”).

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 RUNNING state. In such a case the task that was in RUNNING state up to that time is said to have been preempted by the task that goes to RUNNING state anew. 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 RUNNING state.

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

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

Task state transitions in a typical implementation are shown in Figure 2.1, “Task State Transition Diagram”. Depending on the implementation, there may be other states besides those shown here.

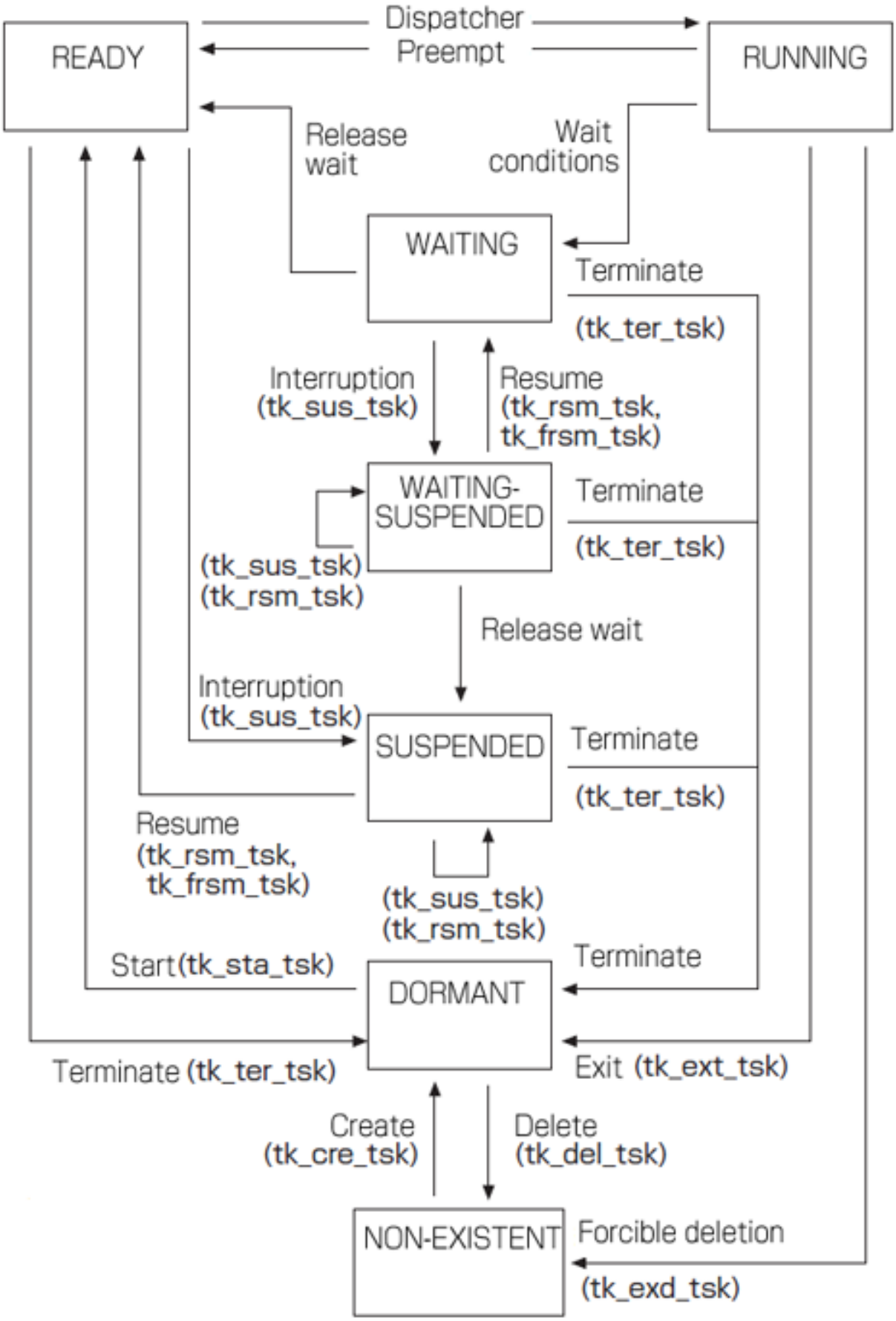

Figure 2.1: Task State Transition Diagram

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 2.1, "State Transitions Distinguishing Invoking Task and Other Tasks"). The reason for this is to clarify task state transitions and facilitate understanding of system calls. This distinction between system call operations in the invoking task and operations affecting other tasks can also be seen as a distinction between state transitions from RUNNING state and those from other states.

	Operations in invoking tasks (Transition from RUNNING state)	Operations on other tasks (Transitions from other states)
Task transition to a waiting state (including SUSPENDED)	tk_slp_tsk RUNNING state → WAITING state	tk_sus_tsk READY state, WAITING state → SUSPENDED state, WAITING-SUSPENDED state
Task exit	tk_ext_tsk RUNNING state → DORMANT state	tk_ter_tsk READY state, WAITING state → DORMANT state
Task deletion	tk_exd_tsk RUNNING state → NON-EXISTENT state	tk_del_tsk DORMANT state → NON-EXISTENT state

Table 2.1: State Transitions Distinguishing Invoking Task and Other Tasks

---

Additional Notes

WAITING state and SUSPENDED state are orthogonally related, in that a request for transition to SUSPENDED 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 WAITING state or WAITING-SUSPENDED state. Thus even if transition to SUSPENDED 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 WAITING-SUSPENDED state, the conditions for allocation of the resource do not change but remain the same as before the request to go to SUSPENDED state.

Rationale for the Specification

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

2.2.2 Task Scheduling Rules

The T-Kernel adopts a preemptive priority-based scheduling method based on priority levels assigned to each task. Tasks having the same priority are scheduled on a FCFS (First Come First Served) basis. Specifically, task precedence is used as the task scheduling rule, and precedence among tasks is determined as follows based on the priority of each task. If there are multiple tasks that can be run, the one with the highest precedence goes to RUNNING state and the others go to READY state. In determining precedence among tasks, of those tasks having different priority levels, that with the highest priority has the highest precedence. Among tasks having the same priority, the one that entered a run state (RUNNING state or READY state) first

has the highest precedence. It is possible, however, to use a system call to change the precedence among tasks having the same priority.

When the task with the highest precedence changes from one task to another, a dispatch occurs immediately and the task in RUNNING state is switched. If no dispatch occurs (during execution of a handler, during dispatch disabled state, etc.), however, the switching of the task in RUNNING state is held off until the next dispatch occurs.

Additional Notes
According to the scheduling rules adopted in the T-Kernel, so long as there is a higher precedence task in a run state, a task with lower precedence will simply not run. That is, unless the highest-precedence task goes to WAITING state or for other reason cannot run, other tasks are not run. This is a fundamental difference from TSS (Time Sharing System) scheduling in which multiple tasks are treated equally.
It is possible, however, to issue a system call changing the precedence among tasks having the same priority. An application can use such a system call to realize round-robin scheduling, which is a typical kind of TSS scheduling.
Examples in figures below illustrate how the task that first goes to a run state (RUNNING state or READY state) gains precedence among tasks having the same priority. Figure 2.2, "Precedence in Initial State" shows the precedence 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 RUNNING state.
When Task A exits, Task B with the next-highest precedence goes to RUNNING state (Figure 2.3, "Precedence After Task B Goes To RUNNING State"). 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 with the same priority. In other words, the task precedence reverts to that in Figure 2.2, "Precedence in Initial State".
Next, consider what happens when Task B goes to WAITING state in the conditions in Figure 2.3, "Precedence After Task B Goes To RUNNING State". Since task precedence is defined among tasks that can be run, the precedence among tasks becomes as shown in Figure 2.4, "Precedence After Task B Goes To WAITING State". Thereafter when the Task B waiting state is released, Task B goes to run state after Task C and Task D, and thus assumes the lowest precedence among tasks of the same priority (Figure 2.5, "Precedence After Task B WAITING State Is Released").
Summarizing the above, immediately after a task that goes from READY state to RUNNING state reverts to READY state, it has the highest precedence among tasks of the same priority; but after a task goes from RUNNING state to WAITING state and then the wait is released, its precedence is the lowest among tasks of the same priority.
Note that after a task goes from SUSPENDED state to a run state, it has the lowest precedence among tasks of the same priority. In a virtual memory system, if a task is made to wait for paging by putting the task in SUSPENDED state, in such a system the task precedence changes as a result of a paging wait.

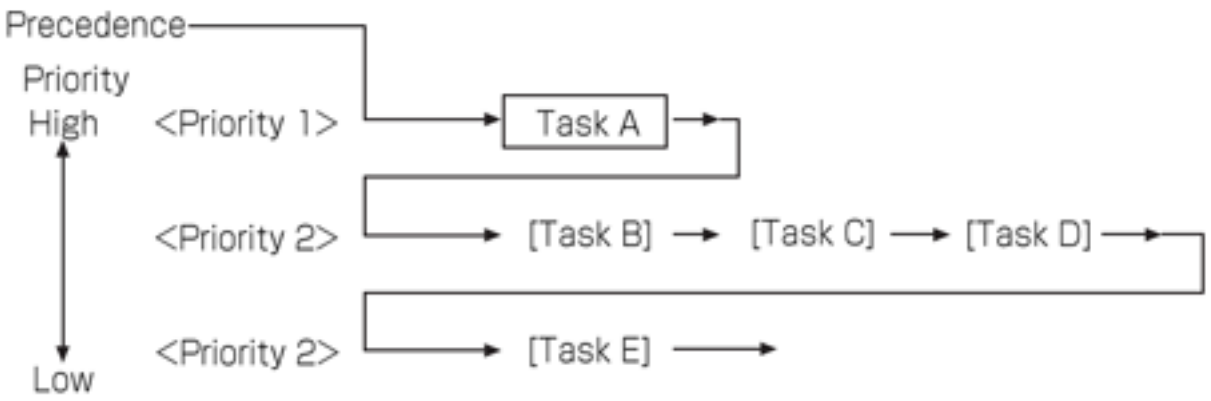

Figure 2.2: Precedence in Initial State

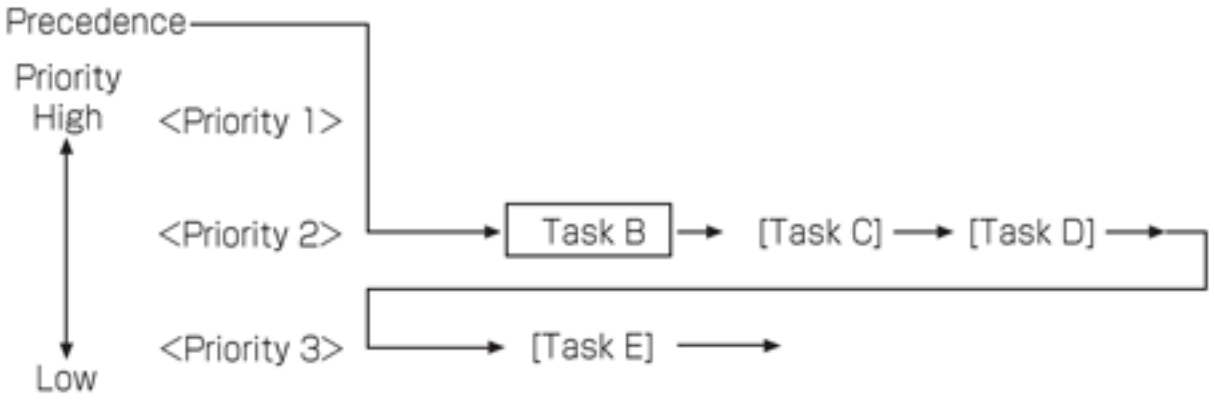

Figure 2.3: Precedence After Task B Goes To RUNNING State

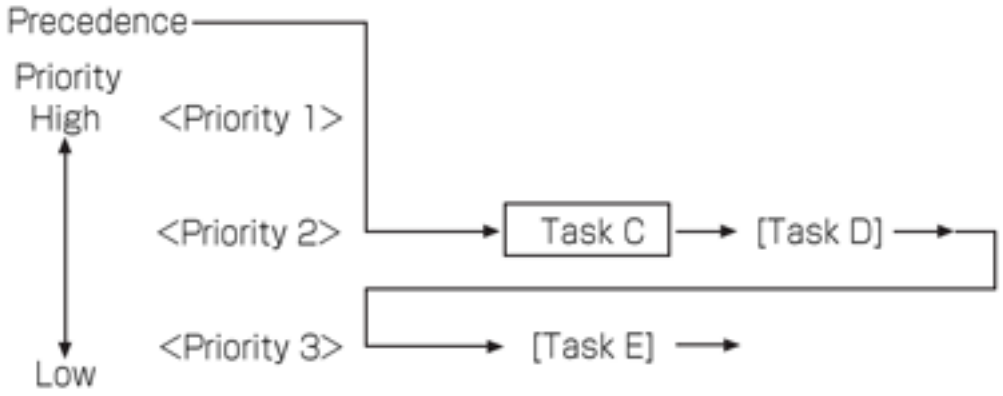

Figure 2.4: Precedence After Task B Goes To WAITING State

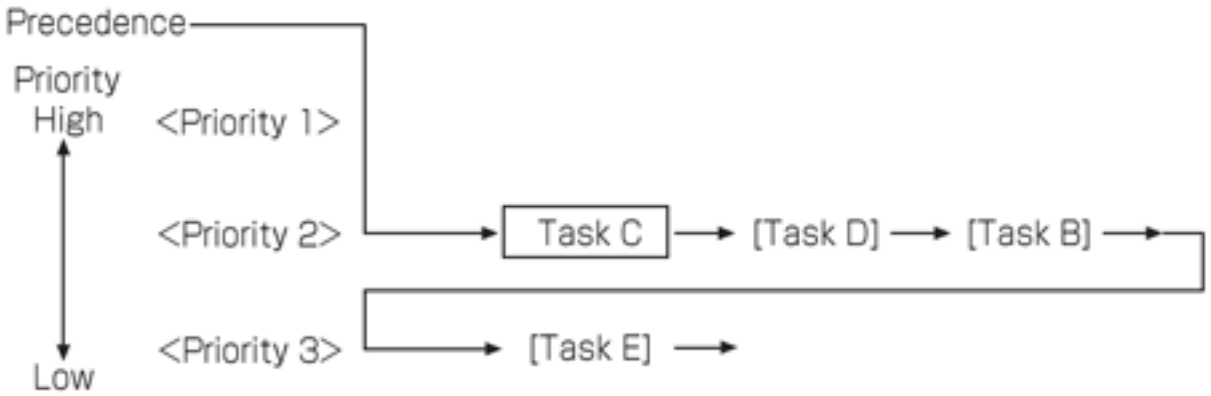

Figure 2.5: Precedence After Task B WAITING State Is Released

2.3 Interrupt Handling

Interrupts in the T-Kernel include both external interrupts from devices and interrupts due to CPU exceptions. One interrupt handler may be defined for each interrupt handler number. Interrupt handlers can be started in two ways: one is to start it without the kernel intervention, the other is to start it via a high-level language support routine.

For more details, see Section 4.8, “Interrupt Management Functions”.

2.4 Task Exception Handling

The T-Kernel defines task exception handling functions for dealing with exceptions. Note that CPU exceptions are treated as interrupts.

A task exception handling function invokes a system call requesting task exception handling by a designated task, interrupts execution by the specified 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 from an application.

For more details, see Section 4.3, “Task Exception Handling Functions”.

2.5 System States

2.5.1 System States While Non-task Portion Is Executing

When programming tasks to run on T-Kernel, one can keep track of the changes in task states by using 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 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 2.6, “Classification of System States”.

Of these shown in Figure 2.6, “Classification of System States”, a "transient state" is equivalent to the kernel running state (system call execution). From the standpoint of the user, it is important that each of the system calls issued by the user application program 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 kernel running is considered a "transient state" and internally 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 in 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 guaranteed. Moreover, task suspension (tk_sus_tsk) may cause a deadlock or other problem by stopping without clearing the transient state.

Accordingly, as a rule tk_ter_tsk and tk_sus_tsk cannot be used in programs. These system calls should be used only in a subsystem such as a virtual memory system or debugger that can be considered to be part of OS.

While being a "non-task portion," the portion that is considered to be running a processing requested from a specific task (called a "requesting task") is called "quasi-task portion." For example, an extended SVC handler in the user-defined subsystem is executed as a "quasi-task portion." The invoking task can be identified in a "quasi-task portion" and the requesting task becomes the invoking task. Similar to the task portion, in the quasi-task portion, the task state transitions can be defined and system calls can be issued to enter into WAITING state from the quasi-task portion. In this way, the quasi-task portion behaves similarly to a subroutine called from a requesting task. "Quasi-task portion" is, however, positioned as an extended part of OS and its processor operation mode and stack space are different from those of the task portion. It means that when a state enters into a quasi-task portion from a task portion, its processor operation mode and stack space are switched. This behavior is different from when a function or subroutine is called in a task portion.

Among the "non-task portion," a "task-independent portion" is activated due to a factor that completely ignore the progress of the task portion or quasi-task portion processing. Specifically, an interrupt handler that is triggered by an external interrupt or a time event handler (cyclic handler and alarm handler) that is triggered due to the specified elapsed time is executed as a "task-independent portion." Note that both the external interrupt and the specified elapsed time are the factors that is independent from a task that is incidentally running at that moment.

Finally, "non-task portion" is separated into three classes: "transient state," "quasi-task portion," and "task-independent portion." The states other than these represent a state where a program for the task is running, this is, the state where "task portion is running."

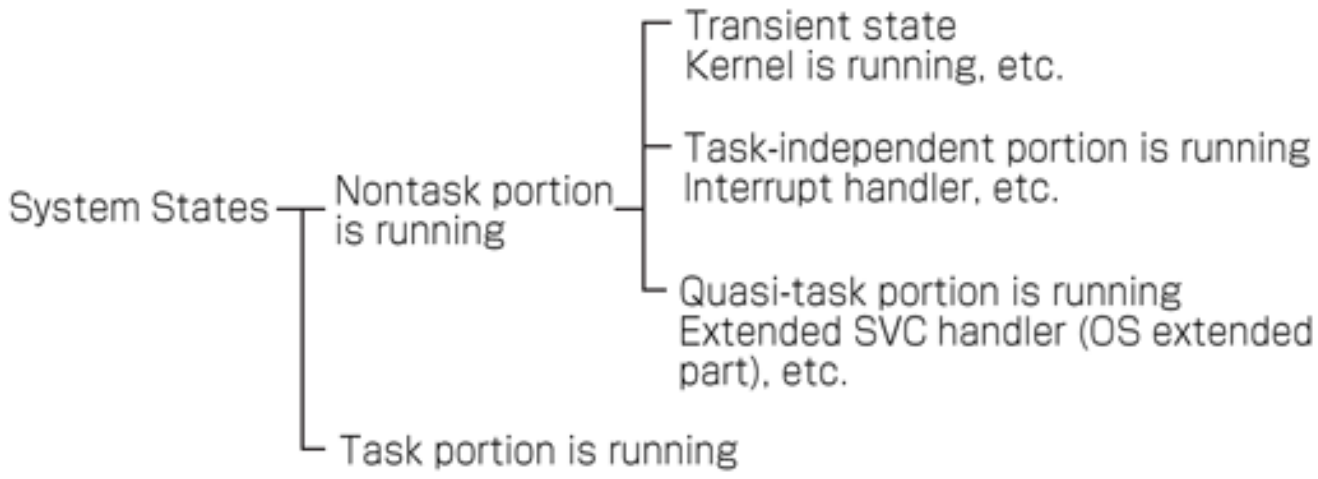

Figure 2.6: Classification of System States

2.5.2 Task-Independent Portion and Quasi-Task Portion

A feature of a task-independent portion (interrupt handlers, time event handlers, etc.) is that it is meaningless to identify the task that was running immediately prior to entering a task-independent portion, and the concept of "invoking task" does not exist. Accordingly, a system call that enters WAITING state, or one that is issued implicitly specifying the invoking task, cannot be called from a task-independent portion. Moreover, since the currently running task cannot be identified in a task-independent portion, there is no task switching (dispatching). If dispatching is necessary, it is delayed until processing leaves the task-independent portion. This is called delayed dispatching.

If dispatching were to take place in the interrupt handler, which is a task-independent portion, the rest of the interrupt handler routine would be delayed for execution after the task started by the dispatching, causing problems in case of interrupt nesting. This is illustrated in Figure 2.7, "Interrupt Nesting and Delayed Dispatching".

In Figure 2.7, "Interrupt Nesting and Delayed Dispatching", Interrupt X is raised during Task A execution, and while its interrupt handler is running, a higher-priority interrupt Y is raised. In this case, if dispatching were to occur immediately on return from interrupt Y at (1),[1] starting Task B, the processing of parts (2) to (3) of Interrupt X would be put off until after Task B relinquishes CPU, with parts (2) to (3) executed only after Task A goes to RUNNING state. The danger is that the low-priority Interrupt X handler would be preempted not only by a higher-priority interrupt but even by Task B started by that interrupt. There would no longer be any guarantee of the interrupt handler execution maintaining priority over task execution, making it impossible to write an interrupt handler. This is the reason for introducing the principle of delayed dispatching.

A feature of a quasi-task portion, on the other hand, is that the task executing prior to entering the quasi-task portion (the requesting task) can be identified, making it possible to define task states just as in the task portion; moreover, it is possible to enter WAITING state while in a quasi-task portion. Accordingly, dispatching occurs in a quasi-task portion in the same way as in ordinary task execution. As a result, even though the OS extended part and other quasi-task portion is a non-task portion, its execution does not necessarily have priority at all times over the task portion. This is in contrast to interrupt handlers, which must always be given execution precedence over tasks.

The following two examples illustrate the difference between a task-independent portion and quasi-task portion.

[1] If dispatching takes place at (1), the remainder of the handler routine for Interrupt X ((2) to (3)) ends up being put off until later.

- An interrupt is raised while Task A (priority 8 = low) is running, and in its interrupt handler (task-independent portion) tk_wup_tsk is issued for Task B (priority 2 = high). 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 are executed. Only when tk_ret_int is executed at the end of the interrupt handler does dispatching occur, causing Task B to run.
- An extended SVC is executed in Task A (priority 8 = low), and in its extended SVC handler (quasi-task portion), tk_wup_tsk is issued for Task B (priority 2 = high). 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 RUNNING 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 RUNNING state.

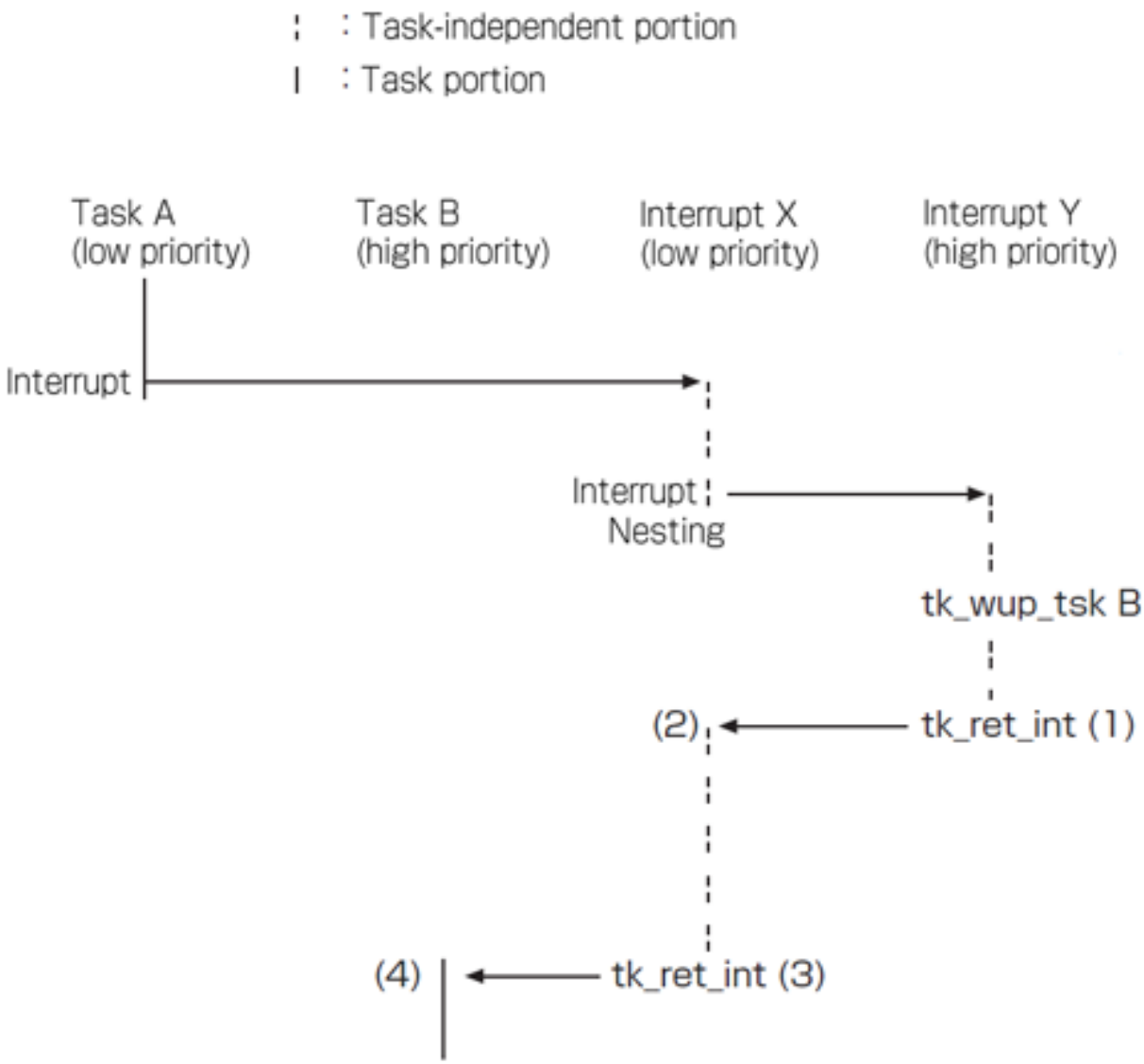

Figure 2.7: Interrupt Nesting and Delayed Dispatching

2.6 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).

Attributes can generally be specified when an object is created. Attributes determine detailed differences in object behavior or the object initial 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 registered.

In an object attribute value, 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. Basically, bits that are not defined in the standard specification can be used as implementation-dependent attributes. 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 registered. Information passed in parameters when an object starts execution has no effect on T-Kernel behavior. Extended information can be read by calling an object status reference system call.

An object is identified by an ID number. In T-Kernel, an ID number is automatically assigned when an object is created. Users cannot specify ID numbers. This makes identifying an object during debugging difficult. We can specify an object name for debugging upon creating each object. This name is used temporarily for debugging and can be referred to only from T-Kernel/DS functions. No check is performed on the naming by T-Kernel.

2.7 Memory

2.7.1 Address Space

Memory address space is divided into system space (shared space) or task space (user space). The system space can be accessed from any task in the same way, and the task space can be accessed only from tasks that belong to that task space [Figure 2.8, “Address Space”]. Multiple tasks may in some cases belong to the same task space.

The logical address space of task space and system space depends on the CPU (and MMU) limitations and is therefore implementation-dependent, but in principle task space should be assigned to low addresses and system space to high addresses.

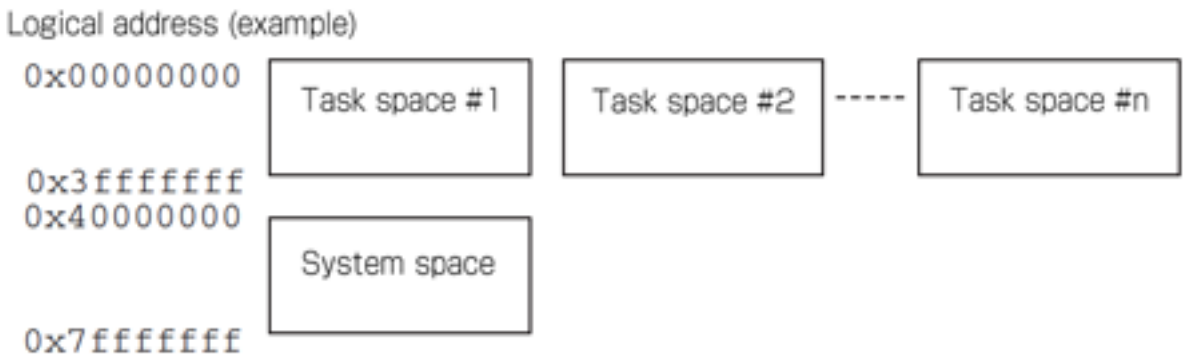

Figure 2.8: Address Space

Since interrupt handlers and other task-independent software are not tasks, they do not have a task space of their own. Instead, while in a task-independent portion they belong to the task executing just before entering the task-independent portion. This is the same as the task space of the currently running task returned by tk_get_tid . When there is no task in RUNNING state, task space is undefined.

As for the system space and task space, other related explanations are available in tk_cre_tsk, Memory Pool Management Functions, and System Memory Management Functions.

In a system with no MMU (or not using an MMU), essentially task space does not exist.

2.7.2 Nonresident Memory

Memory may be resident or nonresident.

When nonresident memory is accessed, data is copied to that memory from a disk or other storage. It therefore requires complicated processing such as disk access by a device driver. 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 OS internal processing, it is necessary to avoid accessing nonresident memory in a critical section. 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 reference nonresident memory is an implementation-dependent matter.

Data transfer from a disk or the like due to nonresident memory access is not performed by T-Kernel. Normally T-Kernel is used along with subsystems that handle virtual memory management and other such processing.

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

2.7.3 Protection Levels

T-Kernel assumes four levels of memory protection, from 0 to 3.

- Level 0 has the highest privilege and level 3 the lowest.
- Access can be made only to memory at the currently running protection level or to levels with lower privilege.
- Changing from one protection level to another is accomplished by invoking a system call or extended SVC, or by interrupt or CPU exception.
- When a protection privilege level of the currently running task is lower than that of the memory being accessed, it is typically the MMU that detects the violation of memory access privilege and raises CPU exception.

The uses of each protection level are as follows.

Protection Levels	Usage
0	Kernel, subsystems, device drivers, etc.
1	System application tasks
2	(reserved)
3	User application tasks

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.

Some MMUs support only two protection levels, privileged and user level. In such a case protection levels 0 to 2 are assigned to privileged level, and protection level 3 to the user level, as if there were 4 levels. In a system with no MMU, all protection levels 0 to 3 are treated as identical.

Chapter 3

Common Rules of T-Kernel

3.1 Data Types

3.1.1 General Data Types

```
typedef signed char          B;     /* signed 8-bit integer */
typedef signed short         H;     /* signed 16-bit integer */
typedef signed long          W;     /* signed 32-bit integer */
typedef signed long long     D;     /* signed 64-bit integer */
typedef unsigned char       UB;     /* unsigned 8-bit integer */
typedef unsigned short      UH;     /* unsigned 16-bit integer */
typedef unsigned long       UW;     /* unsigned 32-bit integer */
typedef unsigned long long  UD;     /* unsigned 64-bit integer */

typedef char                VB;     /* 8-bit data without an intended type */
typedef short               VH;     /* 16-bit data without an intended type */
typedef long                VW;     /* 32-bit data without an intended type */
typedef long long           VD;     /* 64-bit data without an intended type */
typedef void                *VP;    /* pointer to data without an intended type */

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

typedef signed int          INT;    /* signed integer of processor bit width, 32 bits or ↩
    more */
typedef unsigned int        UINT;   /* unsigned integer of processor bit width, 32 bits or ↩
    more */

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

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

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

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

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

Note

- VB, VH, VW, and VD differ from B, H, W, and D in that the former mean only the bit width is known, not the contents of the data type, whereas the latter clearly indicate integer type.
- Processor bit width must be 32 bits or more. INT and UINT must therefore always have a width of 32 bits or more.
- BOOL defines TRUE = 1, but any value other than 0 is also TRUE. For this reason a decision such as bool == TRUE must be avoided. Instead use bool != FALSE.

Additional Notes
Parameters such as stksz, wupcnt, and message size that clearly do not take negative values are also in principle signed integer (INT) 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 enables 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.).

Difference from T-Kernel 1.0

- 64-bit D and UD are added. 'D' means Double integer. "signed" is added to the declaration of a signed integer. int is changed to long to clearly indicate that W and UW are 32-bit.
- Though MSEC in T-Kernel 1.0 was INT (integer with processor bit width), MSEC in T-Kernel 2.0 has been changed to W (integer with 32-bit fixed width). This is a feedback from μT-Kernel specification. That specification was negatively affected when INT is 16-bit was changed to W (integer with 32-bit fixed width).

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 Codes */
typedef INT             RNO;            /* rendezvous number */
typedef UW              ATR;            /* Object/handler attributes */
typedef INT             ER;             /* Error Code */
typedef INT             PRI;            /* Priority */
typedef W               TMO;            /* Timeout specification in milliseconds */
typedef D               TMO_U;          /* Timeout specification in microseconds with 64- ↩
    bit integer */
typedef UW              RELTIM;         /* Relative time in milliseconds */
typedef UD              RELTIM_U;       /* Relative time in microseconds with 64-bit  ↩
    integer */

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

typedef D               SYSTIM_U;       /* System time in microseconds with 64-bit integer  ↩
    */

/*
 * 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 */
```

Note

- 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.

Difference from T-Kernel 1.0

TMO_U that represents timeout specification in microseconds with 64-bit integer, RELTIM_U that represents relative time in microseconds with 64-bit integer, and SYSTIM_U that represents system time in microseconds with 64-bit integer are added. RELTIM_U is unsigned corresponding to RELTIM, and SYSTIM_U is signed corresponding to SYSTIM. Though SYSTIM is a structure comprising two 32-bit members, SYSTIM_U is a plain 64-bit integer rather than a structure to directly take advantage of the convenience of a 64-bit data.

Though TMO that represents timeout specification in milliseconds was INT in T-Kernel 1.0, it has been changed to W in T-Kernel 2.0. Additionally, though ATR that represents an object attribute and others and RELTIM that represents a relative time in milliseconds were UINT in T-Kernel 1.0, they have been changed to UW in T-Kernel 2.0.

Additional Notes

The policy is to append "_u" (u means μ) or "_U" at the end for parameters and data types representing microsecond (μsec), or append "_d" (d means double integer) or "_D" at the end for other parameters and data types representing 64-bit integer. TMO_U, RELTIM_U, and SYSTIM_U are data type names complying to this policy.

3.2 System Calls

3.2.1 System Call Format

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

The method for interfacing with the assembly language shall be implementation-dependent. Calling by means of a C language interface is recommended even when an assembly language is used. In this way, portability is assured for programs written in assembly language even if the OS changes, so long as the CPU is the same.

The following common rules are established for system call interfaces.

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

The processing part (a part in which T-Kernel functions are actually called from within a function that represents a system call) of the system call interface is implemented as a library written in assembly language. This is called an interface library. In consideration of portability, C language macros, in-line functions, in-line assembly codes, etc. are not used for implementation of the interface library.

Among C language interfaces for system calls, those which pass parameters using a packet or pointer have CONST modifier attached to explicitly indicate that T-Kernel does not overwrite a parameter referred to by the pointer.

CONST is intended to be the C language `const` modifier equivalent. This alias for `const` is used so that the compiler check can be disabled by using #define macro function when any program that does not support `const` modifier mixes in.

Specific usage of CONST is as follows: Details, however, depend on the development environment.

1. Include the following descriptions in the common include file:

```
/* If TKERNEL_CHECK_CONST definition exists, enable the check for const */
#ifdef TKERNEL_CHECK_CONST
#define CONST const
#else
#define CONST
#endif
```

2. Describe a function definition or system call definition in the program by using CONST.

Example 3.1 Description Example of CONST

```
tk_cre_tsk( CONST T_CTSK *pk_ctsk );
foo_bar( CONST void *buf );
```

3. Enable `const` by the specification in `Makefile`. (Recommended)

Example 3.2 Example of Enabling `const`

```
CFLAGS += -DTKERNEL_CHECK_CONST
```

※ If the above specification does not exist, the check for `const` is being disabled.

In T-Kernel 2.0 or later, it is strongly recommended that CONST is used explicit by in a program and the check for `const` is enabled in development.

Difference from T-Kernel 1.0
CONST is added to the C language interface of system calls, and the check using `const` modifier is recommended. However, at the same time, the workaround for programs that do not support `const` modifier is also established.

3.2.2 System Calls Possible from Task-Independent Portion

The following system calls can be issued from a task-independent portion and in dispatch disabled state:

System call name	Summary description
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_sig_tev	Signal Task Event
tk_rot_rdq	Rotate Ready Queue
tk_get_tid	Get Task Identifier
tk_sta_cyc	Start Cyclic Handler
tk_stp_cyc	Stop Cyclic Handler
tk_sta_alm	Start Alarm Handler
tk_sta_alm_u	Start Alarm Handler (in microseconds)
tk_stp_alm	Stop Alarm Handler
tk_ref_tsk	Reference Task Status
tk_ref_tsk_u	Reference Task Status (Microseconds)
tk_ref_cyc	Reference Cyclic Handler Status
tk_ref_cyc_u	Reference Cyclic Handler Status (Microseconds)
tk_ref_alm	Reference Alarm Handler Status
tk_ref_alm_u	Reference Alarm Handler Status (Microseconds)
tk_ref_sys	Reference System Status
tk_ret_int	Return from Interrupt Handler (can be issued only from an interrupt handler written in an assembly language)

Whether system calls other than those above can be issued from a task-independent portion or in dispatch disabled state is implementation-dependent.

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 privilege than the specified protection level, the error code E_OACV is returned.

Extended SVC calling cannot be restricted.

If, for example,issuing a system call from a level with lower privilege than level 1 is prohibited, system calls cannot be made 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 that use the functions of T-Kernel extension 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 Section 5.7, “System Configuration Information Management Functions”).

3.2.4 Modifying a Parameter Packet Format

Some parameters passed to system calls use 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 can be added to a parameter packet. It is not allowable, however, to change the data types and order of information defined in the standard specification or to delete any of this information. When implementation-dependent information is added, it must be positioned 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 content is 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 content is indeterminate) and the default values are to be used instead.

The reason for this specification is to ensure that a program developed within the scope of the standard specification will be able to run 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.

The system call function codes are not specified here but are to be defined in implementation.

See tk_def_ssy on extended SVC function codes.

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 values in the case of normal completion is specified individually 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 interface as having no return code (i.e., 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. Main error codes are classified into error classes based on the necessity of their detection, the circumstances in which they occur and other factors. Since T-Kernel/OS does not use a sub error code, these bits are always 0.

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

3.2.7 Timeout

A system call that may enter WAITING 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 error code E_TMOUT.

In accordance with the principle that there should be no side-effects from calling a system call if that system call returns an error code, the calling 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 even when a situation arises in which it would ordinarily go to WAITING state. In other words, a system call with timeout set to 0 when it is invoked has no possibility of entering WAITING state. Invoking a system call with timeout set to 0 is called polling; i.e., a system call that performs polling has no chance of entering WAITING 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 states that the system call "enters WAITING state" or "is put in WAITING state," if a timeout is set and that time interval elapses before processing is completed, the WAITING 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 WAITING state.

Timeout (TMO and TMO_U types) is given as a positive integer, or as `TMO_POL` (= 0) for polling, or as `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 WAITING 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 timer interrupt (sometimes called "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 timer interrupt 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 or RELTIM_U type) is used. If relative time is used to specify event occurrence time, it is necessary to guarantee that the event processing will take place after the specified time has elapsed from the time base. Relative time (RELTIM or RELTIM_U type) is also used for e.g. event occurrence. In such cases the method of interpreting the specified relative time is determined for each case. When time is specified as an absolute value, system time (SYSTIM or SYSTIM_U type) is used. The T-Kernel 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 occurs that was specified using relative time. What changes is the system time at which an event occurs that was specified as relative time.

SYSTIM: System time
Time base 1 millisecond, 64-bit signed integer

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

SYSTIM_U: System time
Time base 1 microsecond, 64-bit signed integer

```
typedef D       SYSTIM_U;       /* 64-bit */
```

RELTIM: Relative time
Time base 1 millisecond, 32-bit unsigned integer (UW)

```
typedef UW      RELTIM;
```

RELTIM_U: Relative time
Time base 1 microsecond, 64-bit unsigned (UD) integer

```
typedef UD      RELTIM_U;       /* Relative time in microseconds with 64-bit integer */
```

TMO: Timeout time
Time base 1 millisecond, 32-bit signed integer (W)

```
typedef W       TMO;
```

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

TMO_U timeout period
Time base 1 microsecond, 64-bit signed (D) integer

```
typedef D       TMO_U;          /* Timeout in microseconds with 64-bit integer */
```

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

Additional Notes
Timeout or other such processing must be guaranteed to occur after the time specified as RELTIM, RELTIM_U, TMO, or TMO_U has elapsed. For example, if the timer interrupt interval is 1 ms and a timeout of 1 ms is specified, timeout occurs on the second timer interrupt after system call invocation. (The first timer interrupt does not exceed 1 ms.)
When a system time (SYSTIM_U) value that may overflow internally in kernel is specified as an argument, the system call behavior is undefined.

3.2.9 Timer Interrupt Interval

Actual time resolution in T-Kernel time management functions uses one that is specified by the "timer interrupt interval" (TTimPeriod) in Section 5.7.2, “Standard System Configuration Information”. By default, the "timer interrupt interval" (TTimPeriod) is set to 10 milliseconds. Actually settable range and operable range are implementation-dependent.

As the "timer interrupt interval" decreases, system overhead by the timer interrupt increases and a clock error may increase due to the constraints on the clock or hardware provided for the timer.

3.3 High-Level Language Support Routines

High-level language support routine capability is provided so that even if a task or handler is written in high-level language, the kernel-related processing can be kept separate from the language environment-related processing. Whether or not a high-level language support routine is used is specified in `TA_HLNG`, one of the object attributes and handler attributes.

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 kernel, the task start address or handler address is a parameter given 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 routines 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 (explicit `return` or "}").

In a system that uses an MMU, however, whereas it is relatively easy to realize 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 kernel, it is more difficult in the case of a task or task exception handler running at a different protection level from the kernel's. 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. Returning a task function using `return` or "}" leads to an undefined behavior. At the end of a task, Exit Task (tk_ext_tsk) or Exit and Delete Task (tk_exd_tsk) must always be issued.

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 3.1, “Behavior of High-Level Language Support Routine”.

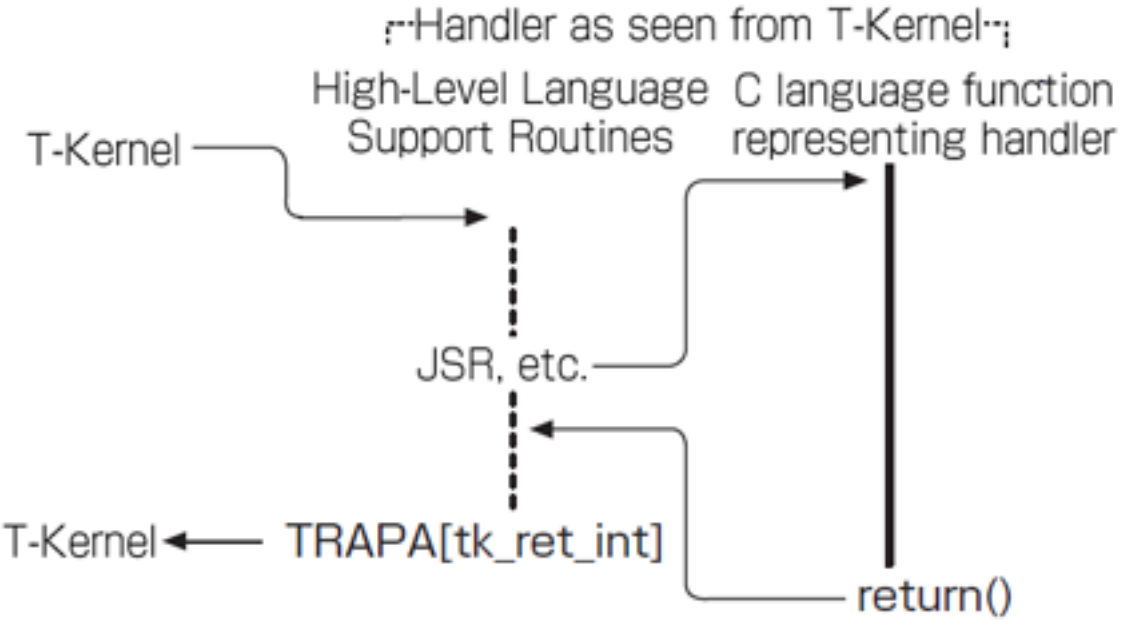

Figure 3.1: Behavior of High-Level Language Support Routine

Chapter 4

T-Kernel/OS Functions

This chapter describes details of the system calls provided by T-Kernel/OS (Operating System).

4.1 Task Management Functions

Task management functions are functions that directly manipulate or reference task states. Functions are provided for creating and deleting a task, for task starting and exit, changing task priority, and referencing task state. A task is an object identified by an ID number called a task ID. Task states and scheduling rules are explained in Section 2.2, “Task States and Scheduling Rules”.

For control of execution order, a task has a base priority and current priority. When simply "task priority" is mentioned, this means the current priority. The base priority of a task is initialized to 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 Section 4.5.1, “Mutex”.

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

4.1.1 tk_cre_tsk - Create Task

C Language Interface

```
#include <tk/tkernel.h>
```

ID tskid = tk_cre_tsk (CONST T_CTSK *pk_ctsk);

Parameter

CONST T_CTSK*	pk_ctsk	Packet to Create Task	Information about task creation

pk_ctsk Detail:

void*	exinf	Extended Information	Extended information
ATR	tskatr	Task Attribute	Task attribute
FP	task	Task Start Address	Task start address
PRI	itskpri	Initial Task Priority	Initial task priority
INT	stksz	Stack Size	Stack size (in bytes)
INT	sstksz	System Stack Size	System stack size (in bytes)
void*	stkptr	User Stack Pointer	User stack pointer
void*	uatb	Address of Task Space Page Table	Task space page table
INT	lsid	Logical Space ID	Logical space ID
ID	resid	Resource ID	Resource ID
UB	dsname[8]	DS Object name	DS object name

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

Return Parameter

ID	tskid	Task ID	Task ID
	or	Error Code	Error code

Error Code

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 coprocessor does not exist
E_NOSPT	Unsupported function (when TA_USERSTACK or TA_TASKSPACE is not supported)
E_PAR	Parameter error
E_ID	Invalid resource ID (resid)
E_NOCOP	The specified coprocessor cannot be used (not installed, or abnormal operation detected)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

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

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

`itskpri` specifies the initial priority at the time the task is started. Task priority values are specified from 1 to 140, with the 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 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, 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`.

`tskatr` indicates system attributes in its lower bits and implementation-dependent attributes in its higher bits. The system attribute part of `tskatr` is as follows.

```
tskatr := (TA_ASM || TA_HLNG)
   | [TA_SSTKSZ] | [TA_USERSTACK] | [TA_TASKSPACE] | [TA_RESID] | [TA_DSNAME]
   | (TA_RNG0 || TA_RNG1 || TA_RNG2 || TA_RNG3)
   | [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_DSNAME	Specifies DS object name
TA_RNGn	Indicates that the task runs at protection level n
TA_COPn	Specifies use of the nth coprocessor (including floating point coprocessor or DSP)
TA_FPU	Specifies use of a floating point coprocessor (when a coprocessor 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. One use of the remaining system attribute fields is for indicating multiprocessor attributes in the future.

```
#define TA_ASM          0x00000000      /* Task in Assembly Language */
#define TA_HLNG         0x00000001      /* Task in High-level language */
#define TA_SSTKSZ       0x00000002      /* System stack size */
#define TA_USERSTACK    0x00000004      /* User stack pointer */
#define TA_TASKSPACE    0x00000008      /* Task space */
#define TA_RESID        0x00000010      /* Task resource group */
#define TA_DSNAME       0x00000040      /* DS 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 */
```

When `TA_HLNG` is specified, starting the `task` jumps to the `task` address not directly but by going 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, void *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 in 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 applies.

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. E_PAR occurs if `TA_RNG0` and `TA_USERSTACK` are specified at the same time.

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. Irrespective of `TA_TASKSPACE` specification, 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 its resource group (see Section 4.10, "Subsystem Management Functions") is specified as 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_DSNAME` is specified, `dsname` is valid and specifies the DS object name. DS object name is used to identify objects by debugger, and it is handled only by T-Kernel/DS API, td_ref_dsname and td_set_dsname. For more details, see the description of td_ref_dsname and td_set_dsname. If `TA_DSNAME` is not specified, `dsname` is ignored. Then td_ref_dsname and td_set_dsname return E_OBJ error.

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 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 system space resident memory used at protection level 0. If TA_USERSTACK is not specified, the user stack is system space resident memory 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 coprocessor. If, for example, the floating point coprocessor is TA_COP0, then TA_FPU = TA_COP0. If there is no particular need to specify the use of a coprocessor 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 implementations where these cannot be supported without an MMU.

4.1.2 tk_del_tsk - Delete Task

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_del_tsk (ID tskid);

Parameter

ID	tskid	Task ID	Task ID

Return Parameter

ER	ercd	Error Code	Error Code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 since 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.

4.1.3 tk_sta_tsk - Start Task

C Language Interface

```
#include <tk/tkernel.h>
```

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

Parameter

ID	tskid	Task ID	Task ID
INT	stacd	Task Start Code	Task start code

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

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.

4.1.4 tk_ext_tsk - Exit Task

C Language Interface

```
#include <tk/tkernel.h>
```

void tk_ext_tsk (void);

Parameter

None

Return Parameter

Does not return to the context issuing the system call.

Error Code

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. The behavior in case an error occurs is implementation-dependent.

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

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 was invoked, leaving the possibility that the program will behave in an unexpected manner. For this reason these system calls do not return even if 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 by tk_cre_tsk at startup. 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_???).

4.1.5 tk_exd_tsk - Exit and Delete Task

C Language Interface

```
#include <tk/tkernel.h>

void tk_exd_tsk ( void );
```

Parameter

None

Return Parameter

Does not return to the context issuing the system call.

Error Code

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. The behavior in case an error occurs is implementation-dependent.

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

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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, 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 was invoked, leaving the possibility that the program will behave in an unexpected manner. For this reason these system calls do not return even if error is detected.

4.1.6 tk_ter_tsk - Terminate Task

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_ter_tsk (ID tskid);

Parameter

ID	tskid	Task ID	Task ID

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 the waiting state (including SUSPENDED state), the waiting 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.

The relationships between target task states and the results of executing tk_ter_tsk are summarized in Table 4.1, “Target Task State and Execution Result (tk_ter_tsk)”.

Additional Notes

When a task is terminated by tk_ter_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 is terminated.

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_ter_tsk, the

Target Task State	tk_ter_tsk `ercd` Return Value	(processing)
Run state (RUNNING or READY) (not for invoking task)	E_OK	Forced termination
Running state (RUNNING) (invoking task)	E_OBJ	No operation
Waiting state (WAITING)	E_OK	Forced termination
Suspended state (SUSPENDED)	E_OK	Forced termination
Waiting-suspended state (WAITING-SUSPENDED)	E_OK	Forced termination
Dormant state (DORMANT)	E_OBJ	No operation
Non-existent state (NON-EXISTENT)	E_NOEXS	No operation

Table 4.1: Target Task State and Execution Result (tk_ter_tsk)

task priority reverts to the startup priority (`itskpri`) that is specified by tk_cre_tsk at startup. The task priority at task termination by tk_ter_tsk is not used after the task is restarted by tk_sta_tsk.

Forcible termination of another task is intended for use only by a debugger or a few other tasks closely related 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 regardless 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 should not be allowed when the task status (what it is executing) is unknown. Ordinary applications therefore must not use the forcible termination function.

4.1.7 tk_chg_pri - Change Task Priority

C Language Interface

```
#include <tk/tkernel.h>
```

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

Parameter

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

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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). However, the priority changed in DORMANT state is valid. The next time the task is started, it has the new initial priority.

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.

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 mutex lock with `TA_INHERIT` 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.

4.1.8 tk_chg_slt - Change Task Slice Time

C Language Interface

```
#include <tk/tkernel.h>
```

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

Parameter

ID	tskid	Task ID	Task ID
RELTIM	slicetime	Slice Time	Slice Time (in ms)

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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 (invalid slicetime)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

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

The slice time 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 tskid = TSK_SELF = 0 is specified in a system call issued from a task-independent portion, error code E_ID is returned.

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

Additional Notes

The time duration 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 regarded as disrupted. In other words, the time duration 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 slice time 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.

4.1.9 tk_chg_slt_u - Change Task Slice Time (in microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_chg_slt_u (ID tskid , RELTIM_U slicetime_u);

Parameter

ID	tskid	Task ID	Task ID
RELTIM_U	slicetime_u	Slice Time	Slice Time (in microseconds)

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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 (invalid slicetime_u)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

This system call takes 64-bit slicetime_u in microseconds instead of the parameter slicetime of tk_chg_slt.

The specification of this system call is same as that of tk_chg_slt, except that the parameter is replaced with slicetime_u. For more details, see the description of tk_chg_slt.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

4.1.10 tk_get_tsp - Get Task Space

C Language Interface

```
#include <tk/tkernel.h>
```

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

Parameter

ID	tskid	Task ID	Task ID
T_TSKSPC*	pk_tskspc	Packet of Task Space	Pointer to the area to return the task space information

Return Parameter

ER	ercd	Error Code	Error code

pk_tskspc Detail:

void*	uatb	Address of Task Space Page Table	Task space page table address
INT	lsid	Logical Space ID	Task space ID (logical space ID)

Error Code

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 (invalid pk_tskspc)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 tskid = TSK_SELF = 0 is specified in a system call issued from a task-independent portion, error code E_ID is returned.

Additional Notes

The precise meaning of pk_tskspc (uatb, lsid) is implementation-dependent, but the above definitions should be followed as much as possible.

4.1.11 tk_set_tsp - Set Task Space

C Language Interface

```
#include <tk/tkernel.h>
```

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

Parameter

ID	tskid	Task ID	Task ID
CONST T_TSKSPC*	pk_tskspc	Packet of Task Space	Task space information

pk_tskspc Detail:

void*	uatb	Address of Task Space Page Table	Task space page table address
INT	lsid	Logical Space ID	Task space ID (logical space ID)

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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 (invalid pk_tskspc)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 tskid = TSK_SELF = 0 is specified in a system call issued from a task-independent portion, error code E_ID is returned.

The kernel is not responsible for handling the side-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.

Additional Notes

The accuracy of pk_tskspc (uatb, lsid) is implementation-dependent, but the above definitions should be followed as much as possible.

4.1.12 tk_get_rid - Refers to resource group to which task belongs

C Language Interface

```
#include <tk/tkernel.h>

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

Parameter

ID	tskid		Task ID	Task ID

Return Parameter

ID	resid		Resource ID	Resource ID
		or	Error Code	Error code

Error Code

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

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 tskid = TSK_SELF = 0 is specified in a system call issued from a task-independent portion, error code E_ID is returned.

Additional Notes

For details of resource group, see Section 4.10, “Subsystem Management Functions”.

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 and tk_del_res).

This system call is used by a subsystem. The subsystem recognizes the process by the resource ID. However, the resource ID cannot be specified when the application issues an extended SVC to make the subsystem. For this reason, the subsystem uses this system call to obtain the resource ID.

4.1.13 tk_set_rid - Set Task Resource ID

C Language Interface

```
#include <tk/tkernel.h>
```

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

Parameter

ID	tskid		Task ID	Task ID
ID	resid		Resource ID	New resource ID

Return Parameter

ID	oldid		Old Resource ID	Old resource ID
		or	Error Code	Error code

Error Code

E_ID	Invalid ID number (tskid or resid is invalid or cannot be used)
E_NOEXS	Object does not exist (the object specified in tskid or resid does not exist)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 tskid = TSK_SELF = 0 is specified in a system call issued from a task-independent portion, error code E_ID is returned.

Additional Notes

For details of resource group, see Section 4.10, “Subsystem Management Functions”.

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.

4.1.14 tk_get_reg - Get Task Registers

C Language Interface

```
#include <tk/tkernel.h>
```

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

Parameter

ID	tskid	Task ID	Task ID
T_REGS*	pk_regs	Packet of Registers	Pointer to the area to return the general register values
T_EIT*	pk_eit	Packet of EIT Registers	Pointer to the area to return the values of registers saved when an exception occurs
T_CREGS*	pk_cregs	Packet of Control Registers	Pointer to the area to return the control register values

Return Parameter

ER	ercd	Error Code	Error code

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

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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.

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 kernel as virtual registers.

4.1.15 tk_set_reg - Set Task Registers

C Language Interface

```
#include <tk/tkernel.h>
```

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

Parameter

ID	tskid	Task ID	Task ID
CONST T_REGS*	pk_regs	Packet of Registers	General registers
CONST T_EIT*	pk_eit	Packet of EIT Registers	Registers saved when EIT occurs
CONST 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 Parameter

ER	ercd	Error Code	Error code

Error Code

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_PAR	Invalid register value (implementation-dependent)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 kernel is not responsible for handling the side-effects of register value changes.

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

If this system call is issued for the invoking task, error code E_OBJ is returned.

4.1.16 tk_get_cpr - Get Task Coprocessor Registers

C Language Interface

```
#include <tk/tkernel.h>
```

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

Parameter

ID	tskid	Task ID	Task ID
INT	copno	Coprocessor Number	Coprocessor number (0 to 3)
T_COPREGS*	pk_copregs	Packet of Coprocessor Registers	Pointer to the area to return coprocessor register values

Return Parameter

ER	ercd	Error Code	Error code

pk_copregs Detail:

T_COP0REG	cop0	Coprocessor Number 0 Register	Coprocessor number 0 register
T_COP1REG	cop1	Coprocessor Number 1 Register	Coprocessor number 1 register
T_COP2REG	cop2	Coprocessor Number 2 Register	Coprocessor number 2 register
T_COP3REG	cop3	Coprocessor Number 3 Register	Coprocessor number 3 register

The contents of T_COPnREG are defined for each CPU and implementation.

Error Code

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_PAR	Parameter error (copno is invalid or the specified coprocessor does not exist)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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.

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 kernel as virtual registers.

4.1.17 tk_set_cpr - Set Task Coprocessor Registers

C Language Interface

```
#include <tk/tkernel.h>
```

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

Parameter

ID	tskid	Task ID	Task ID
INT	copno	Coprocessor Number	Coprocessor number (0 to 3)
CONST T_COPREGS*	pk_copregs	Packet of Coprocessor Registers	Coprocessor register

pk_copregs Detail:

T_COP0REG	cop0	Coprocessor Number 0 Register	Coprocessor number 0 register
T_COP1REG	cop1	Coprocessor Number 1 Register	Coprocessor number 1 register
T_COP2REG	cop2	Coprocessor Number 2 Register	Coprocessor number 2 register
T_COP3REG	cop3	Coprocessor Number 3 Register	Coprocessor number 3 register

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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_PAR	Parameter error (copno is invalid or the specified coprocessor does not exist), or the set register value is invalid (implementation-dependent)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 kernel is not responsible for handling the side-effects of register value changes.

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

If this system call is issued for the invoking task, error code E_OBJ is returned.

4.1.18 tk_inf_tsk - Reference Task Statistics

C Language Interface

```
#include <tk/tkernel.h>
```

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

Parameter

ID	tskid	Task ID	Task ID
T_ITSK*	pk_itsk	Packet to Return Task Statistics	Pointer to the area to return the task statistics
BOOL	clr	Clear	Task statistics clear flag

Return Parameter

ER	ercd	Error Code	Error code

pk_itsk Detail:

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

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

Error Code

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 (invalid pk_itsk)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 tskid = TSK_SELF = 0 is specified in a system call issued from a task-independent portion, error code E_ID is returned.

stime and utime in the task statistics (T_ITSK) return values rounded to milliseconds. To know the value in microseconds, call tk_inf_tsk_u.

Additional Notes

The system-level run time is accumulated while the task runs at TA_RNG0, and the user-level run time is accumulated while the task runs at protection levels other than TA_RNG0. The execution time of 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.

4.1.19 tk_inf_tsk_u - Reference Task Statistics (Microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_inf_tsk_u (ID tskid , T_ITSK_U *pk_itsk_u , BOOL clr);

Parameter

ID	tskid	Task ID	Task ID
T_ITSK_U*	pk_itsk_u	Packet to ReturnTask Statistics	Pointer to the area to return the task statistics
BOOL	clr	Clear	Task statistics clear flag

Return Parameter

ER	ercd	Error Code	Error code

pk_itsk_u Detail:

RELTIM_U	stime_u	System Time	Cumulative system-level run time (in microseconds)
RELTIM_U	utime_u	User Time	Cumulative user-level run time (in microseconds)

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

Error Code

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 (invalid pk_itsk_u)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

This system call takes 64-bit stime_u and utime_u in microseconds instead of the return parameters stime and utime of tk_inf_tsk.

The specification of this system call is same as that of tk_inf_tsk, except that the return parameters are replaced with stime_u and utime_u. For more details, see the description of tk_inf_tsk.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

4.1.20 tk_ref_tsk - Reference Task Status

C Language Interface

```
#include <tk/tkernel.h>
```

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

Parameter

ID	tskid	Task ID	Task ID
T_RTSK*	pk_rtsk	Packet to Return Task Status	Pointer to the area to return the task status

Return Parameter

ER	ercd	Error Code	Error code

pk_rtsk Detail:

void*	exinf	Extended Information	Extended information
PRI	tskpri	Task Priority	Current priority
PRI	tskbpri	Task Base Priority	Base priority
UINT	tskstat	Task State	Task State
UINT	tskwait	Task Wait Factor	Wait factor
ID	wid	Waiting Object ID	Waiting object ID
INT	wupcnt	Wakeup Count	Wakeup request queuing count
INT	suscnt	Suspend Count	Suspend request nesting count
RELTIM	slicetime	Slice Time	Maximum continuous run time (in ms)
UINT	waitmask	Wait Mask	Disabled wait factors
UINT	texmask	Task Exception Mask	Allowed task exceptions
UINT	tskevent	Task Event	Raised task event

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

Error Code

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 (invalid pk_rtsk)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the state of the task specified in tskid.

tskstat takes the following values.

TTS_RUN	0x0001	RUNNING state
TTS_RDY	0x0002	READY state
TTS_WAI	0x0004	WAITING state
TTS_SUS	0x0008	SUSPENDED state
TTS_WAS	0x000c	WAITING-SUSPENDED state
TTS_DMT	0x0010	DORMANT state
TTS_NODISWAI	0x0080	Disabling of wait by tk_dis_wai is prohibited

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 RUNNING 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, TTS_DMT).

In the case of TTS_WAI (including TTS_WAS), disabling of wait by the tk_dis_wai is prohibited, 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, RUNNING (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 4.2, "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 on rendezvous call	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 caused by tk_get_mpf	mpfid
TTW_MPL	0x00004000	Wait caused by 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

Table 4.2: Values of tskwait and wid

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 tskid=TSK_SELF=0 is specified in a system call issued from a task-independent portion, error code E_ID is returned.

When the task specified with tk_ref_tsk does not exist, error code E_NOEXS is returned.

`slicetime` in the task status information (T_RTSK) returns a value rounded to milliseconds. To know the value in microseconds, call tk_ref_tsk_u.

Additional Notes

Even when `tskid` = `TSK_SELF` is specified with 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.

4.1.21 tk_ref_tsk_u - Reference Task Status (Microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_tsk_u (ID tskid , T_RTSK_U *pk_rtsk_u);

Parameter

ID	tskid	Task ID	Task ID
T_RTSK_U*	pk_rtsk_u	Packet to Refer Task Status	Pointer to the area to return the task status

Return Parameter

ER	ercd	Error Code	Error code

pk_rtsk_u Detail:

void*	exinf	Extended Information	Extended information
PRI	tskpri	Task Priority	Current priority
PRI	tskbpri	Task Base Priority	Base priority
UINT	tskstat	Task State	Task State
UINT	tskwait	Task Wait Factor	Wait factor
ID	wid	Waiting Object ID	Waiting object ID
INT	wupcnt	Wakeup Count	Wakeup request queuing count
INT	suscnt	Suspend Count	Suspend request nesting count
RELTIM_U	slicetime_u	Slice Time	Maximum continuous run time (in microseconds)
UINT	waitmask	Wait Mask	Disabled wait factors
UINT	texmask	Task Exception Mask	Allowed task exceptions
UINT	tskevent	Task Event	Raised task event

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

Error Code

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 (invalid pk_rtsk_u)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

This system call takes 64-bit `slicetime_u` in microseconds instead of the return parameter `slicetime` of tk_ref_tsk.

The specification of this system call is same as that of tk_ref_tsk, except that the return parameter is replaced with `slicetime_u`. For more details, see the description of tk_ref_tsk.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

4.2 Task Synchronization Functions

Task 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 WAITING state, for changing a task state to SUSPENDED state, for delaying execution of the invoking task, and for disabling task WAITING state.

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. The 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 SUSPENDED state (including WAITING-SUSPENDED state), the request is remembered, and later when it is attempted to resume the task in SUSPENDED state (including WAITING-SUSPENDED state), it is not resumed. The 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.

4.2.1 tk_slp_tsk - Sleep Task

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_slp_tsk (TMO tmout);

Parameter

TMO	tmout	Timeout	Timeout (ms)

Return Parameter

ER	ercd	Error Code	Error code

Error Codes

E_OK	Normal completion
E_PAR	Parameter error (tmout ≦ (-2))
E_RLWAI	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion, or in dispatch disabled state)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Changes the state of the invoking task from RUNNING state to sleep state (WAITING state for tk_wup_tsk). Note if the wakeup requests for the invoking task are queued, i.e., the wakeup request queuing count of the invoking task is 1 or more, the count is decremented by 1, and the execution is continued without moving the invoking task to the waiting state.

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, the timeout error code E_TMOUT is returned. Specifying tmout = TMO_FEVR (= -1) means eternal wait. 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 the waiting 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 the waiting state by tk_slp_tsk. In this case the task goes to WAITING-SUSPENDED 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 as follows.

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 a task sleeping. In such a situation, normal operation would not be possible in either the application or middleware.

In this manner, proper task synchronization is not possible if 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.

4.2.2 tk_slp_tsk_u - Sleep Task (in microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_slp_tsk_u (TMO_U tmout_u);

Parameter

TMO_U	tmout_u	Timeout	Timeout (in microseconds)

Return Parameter

ER	ercd	Error Code	Error code

Error Codes

E_OK	Normal completion
E_PAR	Parameter error (tmout_u ≦ (-2))
E_RLWAI	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion, or in dispatch disabled state)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

This system call takes 64-bit tmout_u in microseconds instead of the parameter tmout of tk_slp_tsk.

The specification of this system call is same as that of tk_slp_tsk, except that the parameter is replaced with tmout_u. For more details, see the description of tk_slp_tsk.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

4.2.3 tk_wup_tsk - Wakeup Task

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_wup_tsk (ID tskid);

Parameter

ID	tskid	Task ID	Task ID

Return Parameter

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

If the task specified in tskid has been put in WAITING state by tk_slp_tsk, this system call releases the WAITING 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 WAITING state, the wakeup request by tk_wup_tsk is queued. That is, the calling of tk_wup_tsk for the target task is recorded, then when tk_slp_tsk is called after that, the task does not go to WAITING 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 WAITING 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 is not made negative but rather the task goes to WAITING state.

It is always possible to queue tk_wup_tsk at least 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 WAITING state does not return error, but whether an error is returned for the second or subsequent time tk_wup_tsk is called is implementation-dependent.

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

4.2.4 tk_can_wup - Cancel Wakeup Task

C Language Interface

```
#include <tk/tkernel.h>
```

INT wupcnt = tk_can_wup (ID tskid);

Parameter

ID	tskid	Task ID	Task ID

Return Parameter

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Passes in the return value the wakeup request queuing count (wupcnt) for the task specified in tskid, at the same time canceling all wakeup requests. 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 tskid = TSK_SELF = 0 is specified in a system call issued from a task-independent portion, error code E_ID is returned.

Additional Notes

This system call can be used to determine whether the processing was completed within the allotted time when processing is performed that involves cyclic wakeup of a task. Before processing of a prior wakeup request is completed and tk_slp_tsk is called by the waken up task, the task monitoring this task calls tk_can_wup. If wupcnt in the return parameter is 1 or above, this means the previous wakeup request was not processed within the allotted time. Measure can then be taken accordingly to compensate for the delay.

4.2.5 tk_rel_wai - Release Wait

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_rel_wai (ID tskid);

Parameter

ID	tskid	Task ID	Task ID

Return Parameter

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 WAITING state (including when called for the invoking task, or for a task in DORMANT state))

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

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

To the task whose WAITING state was released by tk_rel_wai, the error code E_RLWAI is returned. At this time, the target task is guaranteed to be released from its wait state without the allocation of the waited resource (without the wait release conditions being met).

Wait release requests are not queued by tk_rel_wai. That is, if the task specified in tskid is already in WAITING state, the WAITING state is cleared; but if it is not in WAITING 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 SUSPENDED state. If tk_rel_wai is issued for a task in WAITING-SUSPENDED state, the task goes to SUSPENDED state. If it is necessary to release SUSPENDED state, the separate system call tk_rsm_tsk or tk_frsm_tsk is used.

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 4.3, “Target Task State and Execution Result (tk_rel_wai) ”.

Target Task State	tk_rel_wai `ercd` Return Value	(processing)
Run state (RUNNING or READY) (not for invoking task)	E_OBJ	No operation
Running state (RUNNING) (invoking task)	E_OBJ	No operation
Waiting state (WAITING)	E_OK	Wait released/release wait
Suspended state (SUSPENDED)	E_OBJ	No operation
Waiting-suspended state (WAITING-SUSPENDED)	E_OK	Goes to SUSPENDED state
Dormant state (DORMANT)	E_OBJ	No operation
Non-existent state (NON-EXISTENT)	E_NOEXS	No operation

Table 4.3: Target Task State and Execution Result (tk_rel_wai)

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 WAITING 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 WAITING state effected by tk_slp_tsk, tk_rel_wai releases also WAITING state caused by other factors (tk_wai_flg, tk_wai_sem, tk_rcv_mbx, tk_get_mpl, tk_dly_tsk, etc.).
- Seen from the task in WAITING state, release of the WAITING 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 WAITING state, error code E_OBJ is returned.

4.2.6 tk_sus_tsk - Suspend Task

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_sus_tsk (ID tskid);

Parameter

ID	tskid	Task ID	Task ID

Return Parameter

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 RUNNING state was specified in dispatch disabled state
E_QOVR	Queuing or nesting overflow (too many nested requests in suscnt)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Puts the task specified in tskid in SUSPENDED state and interrupts execution by the task.

SUSPENDED 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 WAITING state, the state goes to a combination of WAITING state and SUSPENDED state (WAITING-SUSPENDED state). Thereafter when the task wait release conditions are met, the task goes to SUSPENDED state. If tk_rsm_tsk is issued for the task in WAITING-SUSPENDED state, the task state reverts to WAITING state (see Figure 2.1, “Task State Transition Diagram”).

Since SUSPENDED 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 this system call is issued from a task-independent portion, if a task in RUNNING state is specified while dispatching is disabled, error code E_CTX is returned.

If tk_sus_tsk is issued more than once for the same task, the task is put in nested SUSPENDED state. 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 of system calls tk_sus_tsk and tk_rsm_tsk is possible.

The nesting feature of suspend requests (issuing tk_sus_tsk two or more times for the same task) and limits on nesting count 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.

Additional Notes

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

In order to delay resource allocation to a task in SUSPENDED state (temporarily lowering its priority), the user can employ 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 page fault processing in a virtual memory system or breakpoint processing in a debugger. As a rule it should not be used in ordinary applications or in middleware. The reason is as follows

task suspension takes place regardless of the target task running state. If, for example, a task is put in SUSPENDED 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 chain reactions, with other tasks stopping and leading to system-wide deadlock.

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

4.2.7 tk_rsm_tsk - Resumes a task in a SUSPENDED state

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_rsm_tsk (ID tskid);

Parameter

ID	tskid	Task ID	Task ID

Return Parameter

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 SUSPENDED state (including when this system call specifies the invoking task or a task in DORMANT state))

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

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

When the target task is in a combined WAITING state and SUSPENDED state (WAITING-SUSPENDED state), executing tk_rsm_tsk releases only the SUSPENDED state, putting the task in WAITING state (see Figure 2.1, "Task State Transition Diagram").

This system call cannot be called 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 SUSPENDED state even after tk_rsm_tsk is executed.

Additional Notes

After a task in RUNNING state or READY state is put in SUSPENDED 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. */
```

4.2.8 tk_frsm_tsk - Force Resume Task

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_frsm_tsk (ID tskid);

Parameter

ID	tskid	Task ID	Task ID

Return Parameter

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 SUSPENDED state (including when this system call specifies the invoking task or a task in DORMANT state))

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

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

When the target task is in a combined WAITING state and SUSPENDED state (WAITING-SUSPENDED state), executing tk_frsm_tsk releases only the SUSPENDED state, putting the task in WAITING state (see Figure 2.1, “Task State Transition Diagram”).

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

Executing tk_frsm_tsk once clears all the nested suspend requests (suscnt) (suscnt = 0). Therefore, all suspend requests are released (suscnt is cleared to 0) even if tk_sus_tsk was issued more than once (suscnt ≧ 2). The SUSPENDED state is always cleared, and unless the task was in the WAITING-SUSPENDED state, its execution resumes.

Additional Notes

After a task in RUNNING state or READY state is put in SUSPENDED 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_frsm_tsk (tskid=task_A);
  /* In this case precedence becomes task_B → task_A. */
```

4.2.9 tk_dly_tsk - Delay Task

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_dly_tsk (RELTIM dlytim);

Parameter

RELTIM	dlytim	Delay Time	Delay time (ms)

Return Parameter

ER	ercd	Error Code	Error code

Error Codes

E_OK	Normal completion
E_PAR	Parameter error (dlytim is invalid)
E_CTX	Context error (issued from task-independent portion, or in dispatch disabled state)
E_RLWAI	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 WAITING state and is subject to release by tk_rel_wai.

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

The time unit 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 specified delay time elapses. 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.

4.2.10 tk_dly_tsk_u - Delay Task (in microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_dly_tsk_u (RELTIM_U dlytim_u);

Parameter

RELTIM_U	dlytim_u	Delay Time	Delay time (microseconds)

Return Parameter

ER	ercd	Error Code	Error code

Error Codes

E_OK	Normal completion
E_PAR	Parameter error (dlytim_u is invalid)
E_CTX	Context error (issued from task-independent portion, or in dispatch disabled state)
E_RLWAI	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

This system call takes 64-bit dlytim_u in microseconds instead of the parameter dlytim of tk_dly_tsk.

The specification of this system call is same as that of tk_dly_tsk, except that the parameter is replaced with dlytim_u. For more details, see the description of tk_dly_tsk.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

4.2.11 tk_sig_tev - Signal Task Event

C Language Interface

```
#include <tk/tkernel.h>
```

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

Parameter

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

Return Parameter

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 in DORMANT state)
E_PAR	Parameter error (tskevt is invalid)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

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 tskid = TSK_SELF = 0 is specified in a system call issued from a task-independent portion, error code E_ID is returned.

Additional Notes

The task event function is used for task synchronization much like tk_slp_tsk and tk_wup_tsk, but differs from the use of these 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.

4.2.12 tk_wai_tev - Wait Task Event

C Language Interface

```
#include <tk/tkernel.h>
```

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

Parameter

INT	waiptn	Wait Event Pattern	Task event pattern
TMO	tmout	Timeout	Timeout (ms)

Return Parameter

INT	tevptn	Task Event Pattern	Task event status when wait released
	or	Error Code	Error code

Error Codes

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

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 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 unit 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 WAITING 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.

4.2.13 tk_wai_tev_u - Wait Task Event (in microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

INT tevptn = tk_wai_tev_u (INT waiptn , TMO_U tmout_u);

Parameter

INT	waiptn	Wait Event Pattern	Task event pattern
TMO_U	tmout_u	Timeout	Timeout (in microseconds)

Return Parameter

INT	tevptn		Task Event Pattern	Task event status when wait released
		or	Error Code	Error Codes

Error Code

E_PAR	Parameter error (waiptn or tmout_u is invalid)
E_RLWAI	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion, or in dispatch disabled state)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

This system call takes 64-bit tmout_u in microseconds instead of the parameter tmout of tk_wai_tev.

The specification of this system call is same as that of tk_wai_tev, except that the parameter is replaced with tmout_u. For more details, see the description of tk_wai_tev.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

4.2.14 tk_dis_wai - Disable Task Wait

C Language Interface

```
#include <tk/tkernel.h>
```

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

Parameter

ID	tskid	Task ID	Task ID
UINT	waitmask	Wait Mask	Task wait disabled setting

Return Parameter

INT	tskwait	Task Wait	Task state after task wait is disabled
	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_PAR	Parameter error (waitmask is invalid)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 send */
#define TTW_RMBF        0x00000200      /* Wait for message buffer receive */
#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 value disabling not the task wait but the calling of an extended SVC. If TTX_SVC has been set when a task attempts to call an extended SVC, E_DISWAI is returned without calling the extended SVC. This value does not have the effect of terminating an already called extended SVC.

The return value (tskwait) includes the waiting state of a task after the waiting states are disabled by tk_dis_wai. This value is same as tskwait returned by tk_ref_tsk. Information concerning TTX_SVC is not returned in tskwait. A tskwait value of 0 means the task has not entered WAITING state (or the wait was released). If tskwait is not 0, this means the task is in WAITING 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 WAITING state after this system call has taken effect, E_DISWAI is returned.

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

Disabling of wait that is set during an extended SVC 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.

Disabling of wait that is set 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 disabling of wait 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 wait disabled, and rejects any attempt to disable wait by tk_dis_wai.

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

Additional Notes

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

Disabling wait 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 waiting state, then the wait is released.

Some specific examples are given in the following.

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.

Disabling of wait 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, an attempt to disable wait 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 will go to WAITING on rendezvous call as a result of the forwarding. However, wait has been disabled by TTW_CAL. So E_DISWAI is returned to both the rendezvous calling task (tk_cal_por) and forwarding task (tk_fwd_por).

4.2.15 tk_ena_wai - Enable Task Wait

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_ena_wai (ID tskid);

Parameter

ID	tskid	Task ID	Task ID

Return Parameter

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Releases all disabling of waits 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 tskid = TSK_SELF = 0 is specified in a system call issued from a task-independent portion, error code E_ID is returned.

4.3 Task Exception Handling Functions

Task exception handling functions handle exception events that are raised for a task in the context of that task.

The task exception handler is started when all the following processing has taken place:

1. Register task exception handler by tk_def_tex
2. Enable task exception by tk_ena_tex
3. Raise task exception by tk_ras_tex

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 set of system calls are available.

A task exception handler can be started only when the target task is running in a task portion. If the task is running in any other portion when a task exception is raised, the task exception handler is started only after the control 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 are specified by task exception codes from 0 to 31, of which 0 has 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 cannot be executed by nesting them. 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 executed by nesting it 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 by calling tk_ext_tsk or tk_exd_tsk.

4.3.1 tk_def_tex - Define Task Exception Handler

C Language Interface

```
#include <tk/tkernel.h>
```

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

Parameter

ID	tskid	Task ID	Task ID
CONST T_DTEX*	pk_dtex	Packet to Define Task Exception	Task exception handler definition information

pk_dtex Detail:

ATR	texatr	Task Exception Attribute	Task exception handler attributes
FP	texhdr	Task Exception Handler	Task exception handler address

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

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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 is invalid or cannot be used)
E_PAR	Parameter error (pk_dtex is invalid or cannot be used)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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.

texatr indicates system attributes in its lower bits and implementation-dependent attributes in its higher bits. The texatr system attributes are not assigned in the present version of T-Kernel specification, 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 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 kernel vendor must provide the assembly language source code 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.

4.3.2 tk_ena_tex - Enable Task Exception

C Language Interface

```
#include <tk/tkernel.h>
```

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

Parameter

ID	tskid	Task ID	Task ID
UINT	texptn	Task Exception Pattern	Task exception pattern

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Enables 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. If the current exception enabled status is texmask, it changes as follows.

enable: texmask |= texptn

If all the bits of texptn are cleared to 0, no operation is made to texmask. No error will result in this case.

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

This system call can be called to tasks in DORMANT state.

4.3.3 tk_dis_tex - Disable Task Exception

C Language Interface

```
#include <tk/tkernel.h>
```

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

Parameter

ID	tskid	Task ID	Task ID
UINT	texptn	Task Exception Pattern	Task exception pattern

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

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

disable: texmask &= ~texptn

If all the bits of texptn are cleared to 0, no operation is made to texmask. No error will result in either case.

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).

This system call can be called to tasks in DORMANT state.

4.3.4 tk_ras_tex - Raise Task Exception

C Language Interface

```
#include <tk/tkernel.h>
```

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

Parameter

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

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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 (the task specified in tskid is 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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Raises the task exception specified in texcd for the task specified in tskid. If the task specified in tskid disables the task exception specified in texcd, the raised task exception is ignored, and is not made pending. In this case, E_OK is returned to this system call.

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 function corresponding to the extended SVC runs as a quasi-task portion of the task that issued tk_ras_tex. That is, it is executed in the context of the quasi-task portion whose requesting task is the task that issued tk_ras_tex.

In such a case tk_ras_tex does not return control until the break function processing ends. For this reason, the specification does not allow tk_ras_tex to be issued from a task-independent portion.

Task exceptions raised in the task that called tk_ras_tex while the break function is running are held until the break function ends.

4.3.5 tk_end_tex - end task exception handler

C Language Interface

```
#include <tk/tkernel.h>
```

INT texcd = tk_end_tex (BOOL enatex);

Parameter

BOOL	enatex	Enable Task Exception	Task exception handler calling enabled flag

Return Parameter

INT	texcd	Task Exception Code	Raised exception code (0 to 31)
	or	Error Code	Error code

Error Code

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

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 in this case. The task must be terminated by calling tk_ext_tsk or tk_exd_tsk. If tk_end_tex is called while processing the task exception code 0, the behavior 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 called and there are pending task exceptions, another task exception handler call is made immediately following tk_end_tex. In this case, 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 ){
    /*
      Processing for task exception 0
    */
    tk_exd_tsk();
  }

  do {
    /*
      Processing for task exception 1～31
    */
  } 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_end_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 independently of executing tasks; so in practice this is not a problem.

4.3.6 tk_ref_tex - Reference Task Exception Status

C Language Interface

```
#include <tk/tkernel.h>
```

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

Parameter

ID	tskid	Task ID	Task ID
T_RTEX*	pk_rtex	Packet to Return Task Exception Status	Pointer to the area to return the task exception status

Return Parameter

ER	ercd	Error Code	Error code

pk_rtex Detail:

UINT	pendtex	Pending Task Exception	Pending task exceptions
UINT	texmask	Task Exception Mask	Allowed task exceptions

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

Error Code

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 (invalid pk_rtex)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 tskid = TSK_SELF = 0 is specified in a system call issued from a task-independent portion, error code E_ID is returned.

4.4 Synchronization and Communication Functions

Synchronization and communication functions use objects independent of tasks used 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. The ID number for the semaphore is 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 be negative), a task attempting to acquire resources goes into WAITING 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. Error is reported if it is attempted to return resources to a semaphore that would cause this maximum count to be exceeded.

4.4.1.1 tk_cre_sem - Create Semaphore

C Language Interface

```
#include <tk/tkernel.h>
```

ID semid = tk_cre_sem (CONST T_CSEM *pk_csem);

Parameter

CONST T_CSEM*	pk_csem	Packet to Create Semaphore	Semaphore creation information

pk_csem Detail:

void*	exinf	Extended Information	Extended information
ATR	sematr	Semaphore Attribute	Semaphore attribute
INT	isemcnt	Initial Semaphore Count	Initial semaphore count
INT	maxsem	Maximum Semaphore Count	Maximum semaphore count
UB	dsname[8]	DS Object name	DS object name

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

Return Parameter

ID	semid	Semaphore ID	Semaphore ID
	or	Error Code	Error code

Error Code

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, or isemcnt or maxsem is negative or invalid)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Creates a semaphore, assigning to it 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 including and above 65536 can be set is implementation-dependent.

exinf can be used freely by the user to set 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 may need 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 kernel pays no attention to the contents of exinf.

sematr indicates system attributes in its lower bits and implementation-dependent attributes in its higher bits. The system attribute part of sematr is as follows.

```
sematr:= (TA_TFIFO || TA_TPRI) | (TA_FIRST || TA_CNT) | [TA_DSNAME] | [TA_NODISWAI]
```

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_DSNAME	Specifies DS object name
TA_NODISWAI	Disabling of wait by tk_dis_wai is prohibited

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 and 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 amount can be allocated receive resources. This is not the same as allocating in order of fewest requests.

When TA_DSNAME is specified, dsname is valid and specifies the DS object name. DS object name is used to identify objects by debugger, and it is handled only by T-Kernel/DS API, td_ref_dsname and td_set_dsname. For more details, see the description of td_ref_dsname and td_set_dsname. If TA_DSNAME is not specified, dsname is ignored. Then td_ref_dsname and td_set_dsname return E_OBJ error.

```
#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_DSNAME       0x00000040      /* DS object name */
#define TA_NODISWAI     0x00000080      /* reject request to disable wait */
```

4.4.1.2 tk_del_sem - Delete Semaphore

C Language Interface

```
#include <tk/tkernel.h>

ER ercd = tk_del_sem (ID semid );
```

Parameter

ID	semid	Semaphore ID	Semaphore ID

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 for condition fulfillment on the semaphore, but error code E_DLT is returned to the task in WAITING state.

4.4.1.3 tk_sig_sem - Signal Semaphore

C Language Interface

```
#include <tk/tkernel.h>
```

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

Parameter

ID	semid	Semaphore ID	Semaphore ID
INT	cnt	Count	Resource return count

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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 over limit)
E_PAR	Parameter error (cnt ≦ 0)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

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 allocated resources goes to READY state. In some conditions more than one task may be allocated resources and put in READY state.

If the semaphore count increases to the point where the maximum count (maxsem) 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.

Additional Notes

Error is not returned even if semcnt goes over the semaphore initial count (isemcnt). When semaphores are used not for mutual exclusion control but for synchronization (like tk_wup_tsk and tk_slp_tsk), the semaphore count (semcnt) will sometimes go over 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 the error that is returned when the count increases.

4.4.1.4 tk_wai_sem - Wait on Semaphore

C Language Interface

```
#include <tk/tkernel.h>
```

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

Parameter

ID	semid	Semaphore ID	Semaphore ID
INT	cnt	Count	Resource request count
TMO	tmout	Timeout	Timeout (ms)

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion, or in dispatch disabled state)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Obtains 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 WAITING state but continues executing. In this case the semaphore count (semcnt) is decreased by the size of cnt. If the resources are not available, the task issuing this system call enters WAITING 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 unit 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 WAITING 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.

4.4.1.5 tk_wai_sem_u - Wait on Semaphore (in microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_wai_sem_u (ID semid , INT cnt , TMO_U tmout_u);

Parameter

ID	semid	Semaphore ID	Semaphore ID
INT	cnt	Count	Resource request count
TMO_U	tmout_u	Timeout	Timeout (in microseconds)

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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_u ≦ (-2), cnt ≦ 0)
E_DLT	The object being waited for was deleted (the specified semaphore was deleted while waiting)
E_RLWAI	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion, or in dispatch disabled state)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

This system call takes 64-bit tmout_u in microseconds instead of the parameter tmout of tk_wai_sem.

The specification of this system call is same as that of tk_wai_sem, except that the parameter is replaced with tmout_u. For more details, see the description of tk_wai_sem.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

4.4.1.6 tk_ref_sem - Reference Semaphore Status

C Language Interface

```
#include <tk/tkernel.h>
```

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

Parameter

ID	semid	Semaphore ID	Semaphore ID
T_RSEM*	pk_rsem	Packet to Return Semaphore Status	Pointer to the area to return the semaphore status

Return Parameter

ER	ercd	Error Code	Error code

pk_rsem Detail:

void*	exinf	Extended Information	Extended information
ID	wtsk	Waiting Task ID	Waiting task ID
INT	semcnt	Semaphore Count	current semaphore count value

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

Error Code

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 (invalid pk_rsem)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

References the status of the semaphore specified in semid, passing in the return parameters the current semaphore count (semcnt), the waiting task ID (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.

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 events. Functions are provided for creating and deleting an event flag, for event flag setting and clearing, event flag waiting, and event flag status reference. An event flag is an object identified by an ID number. The ID number for the event flag is 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 called simply 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.

4.4.2.1 tk_cre_flg - Create Event Flag

C Language Interface

```
#include <tk/tkernel.h>

ID flgid = tk_cre_flg (CONST T_CFLG *pk_cflg );
```

Parameter

CONST T_CFLG*	pk_cflg	Packet to Create EventFlag	Event flag creation information

pk_cflg Detail:

void*	exinf	Extended Information	Extended information
ATR	flgatr	EventFlag Attribute	Event flag attribute
UINT	iflgptn	Initial EventFlag Pattern	Event flag initial value
UB	dsname[8]	DS Object name	DS object name

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

Return Parameter

ID	flgid		EventFlag ID	Event flag ID
		or	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Creates an event flag, assigning to it an event flag ID. This system call allocates a control block to the created event flag and sets its initial value to iflgptn. An event flag handles one word's worth of bits as a group. All operations are performed in single word units.

exinf can be used freely by the user to set 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 may need 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 kernel pays no attention to the contents of exinf.

flgatr indicates system attributes in its lower bits and implementation-dependent attributes in its higher bits. The system attribute part of flgatr is as follows.

```
flgatr:= (TA_TFIFO || TA_TPRI) | (TA_WMUL || TA_WSGL) | [TA_DSNAME] | [TA_NODISWAI]
```

TA_TFIFO	Tasks are queued in FIFO order
TA_TPRI	Tasks are queued in priority order
TA_WSGL	Waiting by multiple tasks is not allowed (Wait Single Task)
TA_WMUL	Waiting by multiple tasks is allowed (Wait Multiple Tasks)
TA_DSNAME	Specifies DS object name
TA_NODISWAI	Disabling of wait by tk_dis_wai is prohibited

When TA_WSGL is specified, multiple tasks cannot be in the WAITING 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 WAITING 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_DSNAME is specified, dsname is valid and specifies the DS object name. DS object name is used to identify objects by debugger, and it is handled only by T-Kernel/DS API, td_ref_dsname and td_set_dsname. For more details, see the description of td_ref_dsname and td_set_dsname. If TA_DSNAME is not specified, dsname is ignored. Then td_ref_dsname and td_set_dsname return E_OBJ error.

```
#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      /* permit multiple task waiting */
#define TA_DSNAME       0x00000040      /* DS object name */
#define TA_NODISWAI     0x00000080      /* reject request to disable wait */
```

4.4.2.2 tk_del_flg - Delete Event Flag

C Language Interface

```
#include <tk/tkernel.h>

ER ercd = tk_del_flg (ID flgid );
```

Parameter

ID	`flgid`	EventFlag ID	Event flag ID

Return Parameter

ER	`ercd`	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 in WAITING state.

4.4.2.3 tk_set_flg - Set Event Flag

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_set_flg (ID flgid , UINT setptn);

Parameter

ID	flgid	EventFlag ID	Event flag ID
UINT	setptn	Set Bit Pattern	Bit pattern to be set

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

tk_set_flg sets the bits indicated in setptn in a one-word event flag specified in flgid. That is, a logical sum is taken of the values of the event flag specified in flgid and the values indicated in setptn. (the processing flgptn |= setptn is executed for the event flag value flgptn)

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 WAITING state of that task is cleared, putting it in RUNNING state or READY state (or SUSPENDED state if the waiting task was in WAITING-SUSPENDED state).

If all the bits of setptn are cleared to 0 in tk_set_flg, no operation is made to the target event flag. 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.

4.4.2.4 tk_clr_flg - Clear Event Flag

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_clr_flg (ID flgid , UINT clrptn);

Parameter

ID	flgid	EventFlag ID	Event flag ID
UINT	clrptn	Clear Bit Pattern	Bit pattern to be cleared

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

tk_clr_flg clears the bits of the one-word event flag specified in flgid, based on the corresponding zero bits of clrptn. That is, a logical product is taken of the values of the event flag specified in flgid and the values indicated in clrptn.(the processing flgptn &= clrptn is executed for the event flag value flgptn)

Issuing tk_clr_flg never results in wait conditions being released for a task waiting for the specified event flag; that is, dispatching never occurs with tk_clr_flg.

If all the bits of clrptn are set to 1 in tk_clr_flg, no operation is made to the target event flag. No error will be returned in either case.

4.4.2.5 tk_wai_flg - Wait Event Flag

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_wai_flg (ID flgid , UINT waiptn , UINT wfmode , UINT *p_flgptn , TMO tmout);

Parameter

ID	`flgid`	EventFlag ID	Event flag ID
UINT	`waiptn`	Wait Bit Pattern	Wait bit pattern
UINT	`wfmode`	Wait EventFlag Mode	Wait release condition
UINT*	`p_flgptn`	Pointer to EventFlag Bit Pattern	Pointer to the area to return the return parameter `flgptn`
TMO	`tmout`	Timeout	Timeout (ms)

Return Parameter

ER	`ercd`	Error Code	Error code
UINT	`flgptn`	EventFlag Bit Pattern	Event flag bit pattern

Error Code

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	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion, or in dispatch disabled state)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 WAITING 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 WAITING 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 WAITING 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 WAITING 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 meets the wait release conditions of this system call. The contents of flgptn are indeterminate if the wait is released due to timeout or the like.

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 unit 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 WAITING 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 WAITING 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. 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.

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 WAITING 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 released from waiting simultaneously 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.

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 by tk_set_flg before a previous message has been read by tk_wai_flg, the previous message will be lost; that is, 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 4.1, "Multiple Tasks Waiting for One Event Flag"). 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.

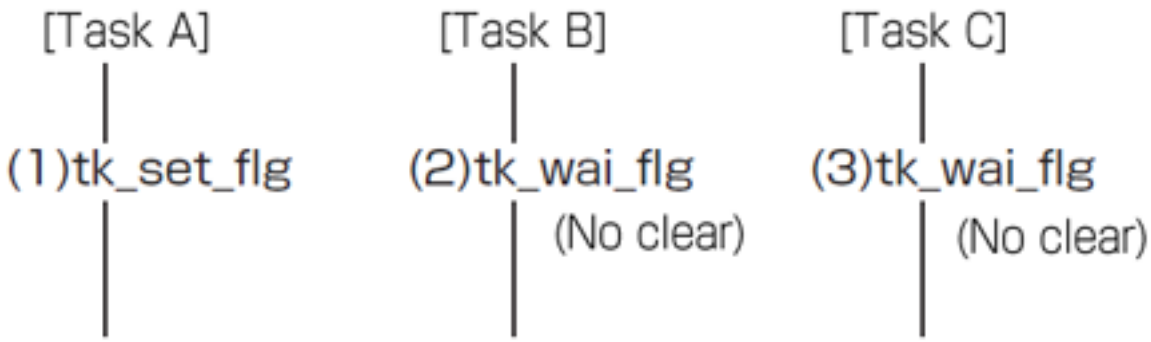

Figure 4.1: Multiple Tasks Waiting for One Event Flag

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 WAITING state regardless of the subsequent event flag values.

4.4.2.6 tk_wai_flg_u - Wait Event Flag (in microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_wai_flg_u (ID flgid , UINT waiptn , UINT wfmode , UINT *p_flgptn , TMO_U tmout_u);

Parameter

ID	`flgid`	EventFlag ID	Event flag ID
UINT	`waiptn`	Wait Bit Pattern	Wait bit pattern
UINT	`wfmode`	Wait EventFlag Mode	Wait mode
UINT*	`p_flgptn`	Pointer to EventFlag Bit Pattern	Pointer to the area to return the return parameter `flgptn`
TMO_U	`tmout_u`	Timeout	Timeout (in microseconds)

Return Parameter

ER	`ercd`	Error Code	Error code
UINT	`flgptn`	EventFlag Bit Pattern	Bit pattern of wait releasing

Error Code

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_u` ≦ (-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	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion, or in dispatch disabled state)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

This system call takes 64-bit `tmout_u` in microseconds instead of the parameter `tmout` of tk_wai_flg.

The specification of this system call is same as that of tk_wai_flg, except that the parameter is replaced with `tmout_u`. For more details, see the description of tk_wai_flg.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

4.4.2.7 tk_ref_flg - Reference Event Flag Status

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_flg (ID flgid , T_RFLG *pk_rflg);

Parameter

ID	flgid	EventFlag ID	Event flag ID
T_RFLG*	pk_rflg	Packet to Return EventFlag Status	Pointer to the area to return the event flag status

Return Parameter

ER	ercd	Error Code	Error code

pk_rflg Detail:

void*	exinf	Extended Information	Extended information
ID	wtsk	Waiting Task ID	Waiting task ID
UINT	flgptn	EventFlag Bit Pattern	The current event flag bit pattern

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

Error Code

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 (invalid pk_rflg)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

References the status of the event flag specified in flgid, passing in the return parameters the current flag pattern (flgptn), waiting task ID (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.

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. The ID number for the mailbox is called a 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 (posting 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 WAITING state for receipt 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 space shared both 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 linked listed. An application program must allocate space at the beginning of a message to be sent, for linked list processing 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. [Figure 4.2, "Format of Messages Using a Mailbox"]

The user puts the message contents not at the beginning of the packet but after the header part (the message contents part in the figure).

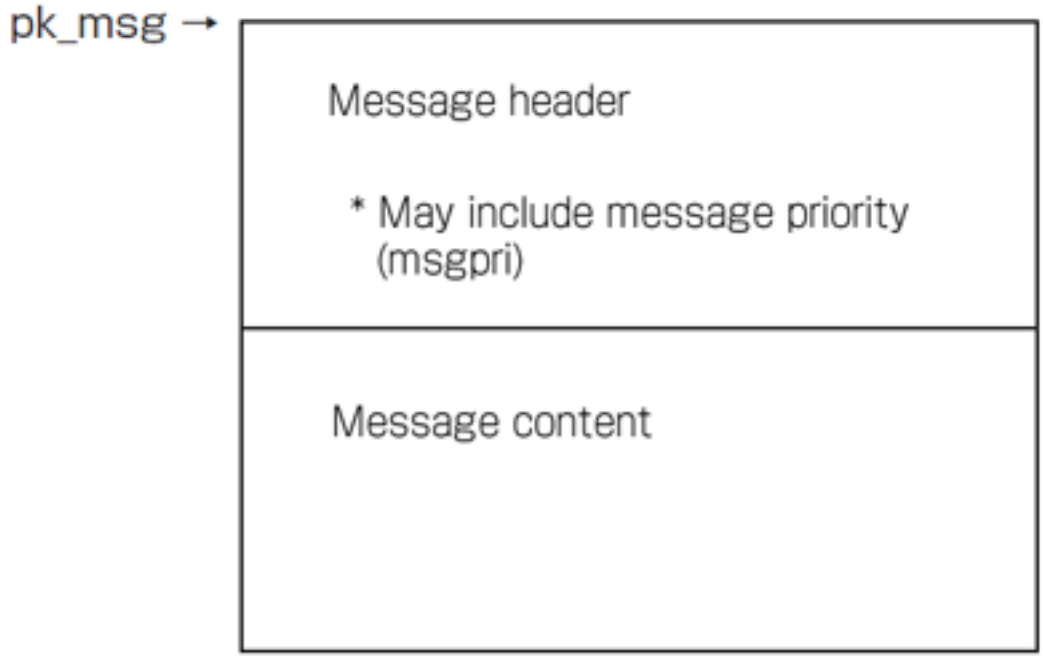

Figure 4.2: Format of Messages Using a Mailbox

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 when an application overwrites the message header is not defined. This specification applies not only to the direct writing of a message header by an application program, but also to the multiple passing of a header address to T-Kernel and having T-Kernel overwrite the message header. Accordingly, the behavior when a message already in the message queue is again sent to a mailbox is undefined.

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 WAITING state.

Memory blocks allocated dynamically from a fixed-size memory pool or variable-size memory pool, or else a statically allocated area can be used for message packets; but these must not be located in task space.

Generally, a sending task allocates a memory block from a memory pool, sending it as a message packet. After a task on the receiving end fetches the message, it returns the memory block directly to its memory pool.

The memory managed by the memory pool management functions is all in system space;

The following sample programs show the above usage:

```
/* Message type definition */
typedef struct  {
  T_MSG msgque;     /* Message header with T_MFIFO attribute */
  UB  msgcont[MSG_SIZE];  /* Message content */
} T_MSG_PACKET;
```

```
/* Task operation that acquires a memory block and sends a message */

  T_MSG_PACKET  *pk_msg;
    ...

  /* Acquire a memory block from the fixed-size memory pool. */
  /* Fixed-memory block size must be sizeof(T_MSG_PACKET) or more */
  tk_get_mpf( mpfid, (void**)&pk_msg, TMO_FEVR );

  /* Create a message at pk_msg -> msgcont[] */
    ...

  /* Send a message */
  tk_snd_mbx( mbxid, (T_MSG*)pk_msg );
```

```
/* Task operation that receives a message and releases a memory block */

  T_MSG_PACKET  *pk_msg;
    ...

  /* Receive a message */
  tk_rcv_mbx( mbxid, (T_MSG**)&pk_msg, TMO_FEVR );

  /* Check message content at pk_msg -> msgcont[] and process them accordingly */
    ...

  /* Return the memory block to the fixed-size memory pool. */
  tk_rel_mpf( mpfid, (void*)pk_msg );
```

4.4.3.1 tk_cre_mbx - Create Mailbox

C Language Interface

```
#include <tk/tkernel.h>
```

ID mbxid = tk_cre_mbx (CONST T_CMBX *pk_cmbx);

Parameter

CONST T_CMBX*	pk_cmbx	Packet to Create Mailbox	Mailbox creation information

pk_cmbx Detail:

void*	exinf	Extended Information	Extended information
ATR	mbxatr	Mailbox Attribute	Mailbox attribute
UB	dsname[8]	DS Object name	DS object name

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

Return Parameter

ID	mbxid	Mailbox ID	Mailbox ID
	or	Error Code	Error code

Error Code

E_NOMEM	Insufficient memory (memory for control block 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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Creates a mailbox, assigning to it a mailbox ID. This system call allocates a control block, etc. for the created mailbox.

exinf can be used freely by the user to set 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 may need 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 kernel pays no attention to the contents of exinf.

mbxatr indicates system attributes in its lower bits and implementation-dependent attributes in its higher bits. The system attribute part of mbxatr is as follows.

```
mbxatr:= (TA_TFIFO || TA_TPRI) | (TA_MFIFO || TA_MPRI) | [TA_DSNAME] | [TA_NODISWAI]
```

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_DSNAME	Specifies DS object name
TA_NODISWAI	Disabling of wait by tk_dis_wai is prohibited

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 priority order. 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_DSNAME is specified, dsname is valid and specifies the DS object name. DS object name is used to identify objects by debugger, and it is handled only by T-Kernel/DS API, td_ref_dsname and td_set_dsname. For more details, see the description of td_ref_dsname and td_set_dsname. If TA_DSNAME is not specified, dsname is ignored. Then td_ref_dsname and td_set_dsname return E_OBJ error.

```
#define TA_TFIFO        0x00000000      /* manage queue by FIFO */
#define TA_TPRI         0x00000001      /* manage queue by priority */
#define TA_MFIFO        0x00000000      /* manage message queue by FIFO */
#define TA_MPRI         0x00000002      /* manage message queue by priority */
#define TA_DSNAME       0x00000040      /* DS object name */
#define TA_NODISWAI     0x00000080      /* reject request to disable wait */
```

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.

4.4.3.2 tk_del_mbx - Delete Mailbox

C Language Interface

```
#include <tk/tkernel.h>

ER ercd = tk_del_mbx (ID mbxid );
```

Parameter

ID	mbxid	Mailbox ID	Mailbox ID

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 in WAITING state. Even if there are messages still in the deleted mailbox, the mailbox is deleted without returning an error code.

4.4.3.3 tk_snd_mbx - Send Message to Mailbox

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_snd_mbx (ID mbxid , T_MSG *pk_msg);

Parameter

ID	mbxid	Mailbox ID	Mailbox ID
T_MSG*	pk_msg	Packet of Message	Start address of message packet

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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 (invalid pk_msg, or msgpri ≦ 0)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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. Therefore, the content of the message packet must not be overwritten until it is fetched by the task that receives this message.

If tasks are already waiting for messages in the same mailbox, the WAITING state of the task at the head of the queue is released, and the pk_msg specified in 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 WAITING state.

pk_msg is the start address of the packet containing the message, including header. The message header has the following format.

```
typedef struct t_msg {
        ?       ?               /* Implementation-dependent content (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). In either case the message header has a fixed-size, which can be obtained by sizeof(T_MSG) or sizeof (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 variable.

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 sending task itself, but the sent message. So while there are queues of waiting messages and receiving tasks, the sending task does not go to WAITING state.

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.

4.4.3.4 tk_rcv_mbx - Receive Message from Mailbox

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_rcv_mbx (ID mbxid , T_MSG **ppk_msg , TMO tmout);

Parameter

ID	mbxid	Mailbox ID	Mailbox ID
T_MSG**	ppk_msg	Pointer to Packet of Message	Pointer to the area to return the return parameter pk_msg
TMO	tmout	Timeout	Timeout (ms)

Return Parameter

ER	ercd	Error Code	Error code
T_MSG*	pk_msg	Packet of Message	Start address of message packet

Error Code

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	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion, or in dispatch disabled state)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

tk_rcv_mbx 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 WAITING 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, passing this 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 (before a message arrives), the system call terminates, returning timeout error code E_TMOUT.

Only positive values can be set in tmout. The time unit 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 WAITING 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.

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).

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.

4.4.3.5 tk_rcv_mbx_u - Receive Message from Mailbox (in microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_rcv_mbx_u (ID mbxid , T_MSG **ppk_msg , TMO_U tmout_u);

Parameter

ID	`mbxid`	Mailbox ID	Mailbox ID
T_MSG**	`ppk_msg`	Pointer to Packet of Message	Pointer to the area to return the return parameter `pk_msg`
TMO_U	`tmout_u`	Timeout	Timeout (in microseconds)

Return Parameter

ER	`ercd`	Error Code	Error code
T_MSG*	`pk_msg`	Packet of Message	Start address of message packet

Error Code

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_u` ≦ (-2))
E_DLT	The object being waited for was deleted (the mailbox was deleted while waiting)
E_RLWAI	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion, or in dispatch disabled state)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

This system call takes 64-bit `tmout_u` in microseconds instead of the parameter `tmout` of tk_rcv_mbx.

The specification of this system call is same as that of tk_rcv_mbx, except that the parameter is replaced with `tmout_u`. For more details, see the description of tk_rcv_mbx.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

4.4.3.6 tk_ref_mbx - Reference Mailbox Status

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_mbx (ID mbxid , T_RMBX *pk_rmbx);

Parameter

ID	mbxid	Mailbox ID	Mailbox ID
T_RMBX*	pk_rmbx	Packet to Refer Mailbox Status	Pointer to the area to return the mailbox status

Return Parameter

ER	ercd	Error Code	Error code

pk_rmbx Detail:

void*	exinf	Extended Information	Extended information
ID	wtsk	Waiting Task ID	Waiting task ID
T_MSG*	pk_msg	Packet of Message	Next message to be received

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

Error Code

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 (invalid pk_rmbx)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 ID (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.

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 functions.

4.5.1 Mutex

A mutex is an object for mutual exclusion control among tasks that use shared resources. Priority inheritance mutexes and priority ceiling mutexes are supported, as a mechanism to prevent the problem of unbounded priority inversion that can occur in mutual exclusion 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. The ID number for the mutex is 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 associated with 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.

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 mutex with `TA_CEILING` attribute is created, a ceiling priority is assigned to it, indicating the base priority of the task having the highest base priority among the tasks that will lock that mutex. If a task having a higher base priority than the ceiling priority of the mutex with `TA_CEILING` attribute 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 having locked a mutex with `TA_CEILING` attribute 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 automatically changing the current priority of a 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.

- Task base priority
- When tasks lock mutexes with `TA_INHERIT` attribute, the current priority of the task having the highest current priority of the tasks waiting for those mutexes.
- When tasks lock mutexes with `TA_CEILING` attribute, the highest ceiling priority of the mutex among those mutexes.

Note that when the current priority of a task waiting for a mutex with `TA_INHERIT` attribute changes as the result of a base priority change brought about by mutex operation or tk_chg_pri, it may become 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 mutex with `TA_INHERIT` attribute, dynamic priority inheritance processing may become necessary also for the task locking that mutex.

The T-Kernel 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 implementation-dependent. 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 mutex with `TA_INHERIT` attribute starts waiting for that mutex.
- When task B is waiting for a mutex with `TA_INHERIT` attribute being locked by another task called A, and if the current priority of B is changed to a higher one than that of task A.
- When a task locks a mutex with `TA_CEILING` attribute 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 changes 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. 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.

Additional Notes

TA_TFIFO attribute mutex or TA_TPRI attribute mutex 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.

4.5.1.1 tk_cre_mtx - Create Mutex

C Language Interface

```
#include <tk/tkernel.h>
```

ID mtxid = tk_cre_mtx (CONST T_CMTX *pk_cmtx);

Parameter

CONST T_CMTX*	pk_cmtx	Packet to Create Mutex	Information about the mutex to be created

pk_cmtx Detail:

void*	exinf	Extended Information	Extended information
ATR	mtxatr	Mutex Attribute	Mutex attributes
PRI	ceilpri	Ceiling Priority of Mutex	Mutex ceiling priority
UB	dsname[8]	DS Object name	DS object name

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

Return Parameter

ID	mtxid		Mutex ID	Mutex ID
		or	Error Code	Error code

Error Code

E_NOMEM	Insufficient memory (memory for control block cannot be allocated)
E_LIMIT	Number of mutexes 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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Creates a mutex, assigning to it a mutex ID. This system call allocates a control block, etc. for the created mutex.

exinf can be used freely by the user to set 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 may need 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 kernel pays no attention to the contents of exinf.

mtxatr indicates system attributes in its lower bits and implementation-dependent attributes in its higher bits. The system attribute part of mtxatr is specified as follows.

```
mtxatr:= (TA_TFIFO || TA_TPRI || TA_INHERIT || TA_CEILING) | [TA_DSNAME] | [TA_NODISWAI]
```

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_DSNAME	Specifies DS object name
TA_NODISWAI	Disabling of wait by tk_dis_wai is prohibited

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, ceilpri is valid and specifies the mutex ceiling priority.

When TA_DSNAME is specified, dsname is valid and specifies the DS object name. DS object name is used to identify objects by debugger, and it is handled only by T-Kernel/DS API, td_ref_dsname and td_set_dsname. For more details, see the description of td_ref_dsname and td_set_dsname. If TA_DSNAME is not specified, dsname is ignored. Then td_ref_dsname and td_set_dsname return E_OBJ error.

```
#define TA_TFIFO        0x00000000      /* manage queue by FIFO */
#define TA_TPRI         0x00000001      /* manage queue by priority */
#define TA_INHERIT      0x00000002      /* priority inheritance protocol */
#define TA_CEILING      0x00000003      /* priority ceiling protocol */
#define TA_DSNAME       0x00000040      /* DS object name */
#define TA_NODISWAI     0x00000080      /* reject request to disable wait */
```

4.5.1.2 tk_del_mtx - Delete Mutex

C Language Interface

```
#include <tk/tkernel.h>

ER ercd = tk_del_mtx (ID mtxid );
```

Parameter

ID	mtxid	Mutex ID	Mutex ID

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 in WAITING state.

When a mutex is deleted, a task locking the mutex will have one fewer locked mutexes. If the mutex to be deleted was a priority inheritance mutex (TA_INHERIT) or priority ceiling mutex (TA_CEILING), then deleting the mutex might change the priority of the task that has locked it.

4.5.1.3 tk_loc_mtx - Lock Mutex

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_loc_mtx (ID mtxid , TMO tmout);

Parameter

ID	mtxid	Mutex ID	Mutex ID
TMO	tmout	Timeout	Timeout (ms)

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait
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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Locks the mutex specified in mtxid. If the mutex can be locked immediately, the task issuing this system call continue executing without entering WAITING state, and the mutex goes to locked status. If the mutex cannot be locked, the task issuing this system call enters WAITING 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 unit 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 WAITING 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 wait to until the resource is locked.

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 (`TA_CEILING`) and the base priority[1] of the invoking task is higher than the ceiling priority of the mutex, error code E_ILUSE (upper priority limit exceeded) is returned.

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 (timeout or other reason), the priority of the task locking that mutex can be lowered to the highest of the following three priorities. Whether this lowering takes place is implementation-dependent.

  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.

[1] 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 that was set when the task was created.

4.5.1.4 tk_loc_mtx_u - Lock Mutex (in microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_loc_mtx_u (ID mtxid , TMO_U tmout_u);

Parameter

ID	`mtxid`	Mutex ID	Mutex ID
TMO_U	`tmout_u`	Timeout	Timeout (in microseconds)

Return Parameter

ER	`ercd`	Error Code	Error code

Error Code

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_u` ≦ (-2))
E_DLT	The object being waited for was deleted (the mutex was deleted while waiting for a lock)
E_RLWAI	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait
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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

This system call takes 64-bit `tmout_u` in microseconds instead of the parameter `tmout` of tk_loc_mtx.

The specification of this system call is same as that of tk_loc_mtx, except that the parameter is replaced with `tmout_u`. For more details, see the description of tk_loc_mtx.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

4.5.1.5 tk_unl_mtx - Unlock Mutex

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_unl_mtx (ID mtxid);

Parameter

ID	mtxid	Mutex ID	Mutex ID

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Unlocks the mutex specified in mtxid.

If there are tasks waiting to lock the mutex, the WAITING 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.

Additional Notes

If the unlocked mutex is a priority inheritance mutex (TA_INHERIT) or priority ceiling mutex (TA_CEILING), 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 the operation above, the invoking task priority is lowered to whichever of the following priority is highest.

a. The highest priority among the current priority of the tasks in the queue of the mutex with the TA_INHERIT attribute locked by the invoking task

b. The highest priority among the ceiling priority of the mutexes with the TA_CEILING attribute locked by the invoking task

c. Base priority of the invoking task

Note that the lowering of priority when locked mutexes 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 by T-Kernel.

4.5.1.6 tk_ref_mtx - Refer Mutex Status

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_mtx (ID mtxid , T_RMTX *pk_rmtx);

Parameter

ID	mtxid	Mutex ID	Mutex ID
T_RMTX*	pk_rmtx	Packet to Return Mutex Status	Pointer to the area to return the mutex status

Return Parameter

ER	ercd	Error Code	Error code

pk_rmtx Detail:

void*	exinf	Extended Information	Extended information
ID	htsk	Locking Task ID	ID of task locking the mutex
ID	wtsk	Lock Waiting Task ID	ID of tasks waiting to lock the mutex

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

Error Code

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 (invalid pk_rmtx)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

References the status of the mutex specified in mtxid, passing in the return parameters the task currently locking the mutex (htsk), tasks waiting to lock the mutex (wtsk), and extended information (exinf).

htsk indicates 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 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 mutex does not exist, error code E_NOEXS 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. The ID number for the message buffer is called a message buffer ID.

A message buffer keeps a queue of tasks waiting to send a message (send queue) and a queue of tasks waiting for receive a message (receive queue). It also has a message buffer space for holding sent messages. The message sender (the side posting event notification) copies a message it wants to send to the message buffer. If there is insufficient space in the message buffer area, the task trying to send 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 receive side (waiting for event notification), one message is fetched from the message buffer. If the message buffer has no messages, the task enters WAITING state until the next message is sent. A task waiting for receiving a message from a message buffer is put in the receive queue of that message buffer.

A synchronous message function 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.

Additional Notes
The message buffer behavior when the size of the message buffer space is set to 0 is explained here using the example in Figure 4.3, "Synchronous Communication by Message Buffer". In this example Task A and Task B run asynchronously.

- If Task A calls tk_snd_mbf first, it goes to WAITING state until Task B calls tk_rcv_mbf. In this case Task A is put in the message buffer send queue [Figure 4.3, "Synchronous Communication by Message Buffer" (a)]
- If Task B calls tk_rcv_mbf first, on the other hand, Task B goes to WAITING state until Task A calls tk_snd_mbf. Task B is put in the message buffer receive queue [Figure 4.3, "Synchronous Communication by Message Buffer" (b)].
- At the point where both Task A has called tk_snd_mbf and Task B has called tk_rcv_mbf, a message is passed from Task A to Task B; Thereafter both tasks enter a run state.

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.

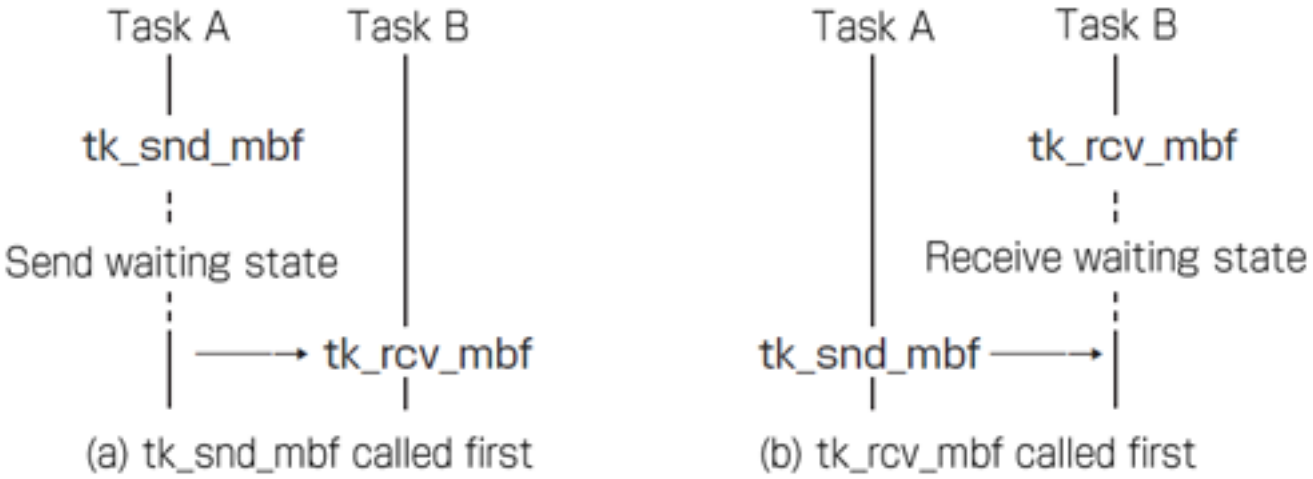

Figure 4.3: Synchronous Communication by Message Buffer

4.5.2.1 tk_cre_mbf - Create Message Buffer

C Language Interface

```
#include <tk/tkernel.h>
```

ID mbfid = tk_cre_mbf (CONST T_CMBF *pk_cmbf);

Parameter

CONST T_CMBF*	pk_cmbf	Packet to Create Message Buffer	Message buffer creation information

pk_cmbf Detail:

void*	exinf	Extended Information	Extended information
ATR	mbfatr	Message Buffer Attribute	Message buffer attribute
INT	bufsz	Buffer Size	Message buffer size (in bytes)
INT	maxmsz	Max Message Size	Maximum message size (in bytes)
UB	dsname[8]	DS Object name	DS object name

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

Return Parameter

ID	mbfid		Message Buffer ID	Message buffer ID
		or	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Creates a message buffer, assigning to it 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. If 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 WAITING state when the buffer is full.

exinf can be used freely by the user to set 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 may need 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 kernel pays no attention to the contents of exinf.

mbfatr indicates system attributes in its lower bits and implementation-dependent attributes in its higher bits. The system attribute part of mbfatr is specified as follows.

```
mbfatr:= (TA_TFIFO || TA_TPRI) | [TA_DSNAME] | [TA_NODISWAI]
```

TA_TFIFO	Tasks waiting on call are queued in FIFO order
TA_TPRI	Tasks waiting on call are queued in priority order
TA_DSNAME	Specifies DS object name
TA_NODISWAI	Disabling of wait by tk_dis_wai is prohibited

The queuing order of tasks waiting for sending a message 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 receiving a message from a message buffer are queued in FIFO order only.

When TA_DSNAME is specified, dsname is valid and specifies the DS object name. DS object name is used to identify objects by debugger, and it is handled only by T-Kernel/DS API, td_ref_dsname and td_set_dsname. For more details, see the description of td_ref_dsname and td_set_dsname. If TA_DSNAME is not specified, dsname is ignored. Then td_ref_dsname and td_set_dsname return E_OBJ error.

```
#define TA_TFIFO        0x00000000      /* manage task queue by FIFO */
#define TA_TPRI         0x00000001      /* manage task queue by priority */
#define TA_DSNAME       0x00000040      /* DS object name */
#define TA_NODISWAI     0x00000080      /* reject request to disable wait */
```

Additional Notes

When there are multiple tasks waiting to send messages, the order in which their 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, the task executing the first system call goes to WAITING state. When the other system call is executed, the message is passed (copied), then both tasks resume running.

In the case of a bufsz = 0 message buffer, the specific functioning is as follows.

1. In Figure 4.4, "Synchronous Communication Using Message Buffer of bufsz = 0", 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 waiting 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 waiting 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.

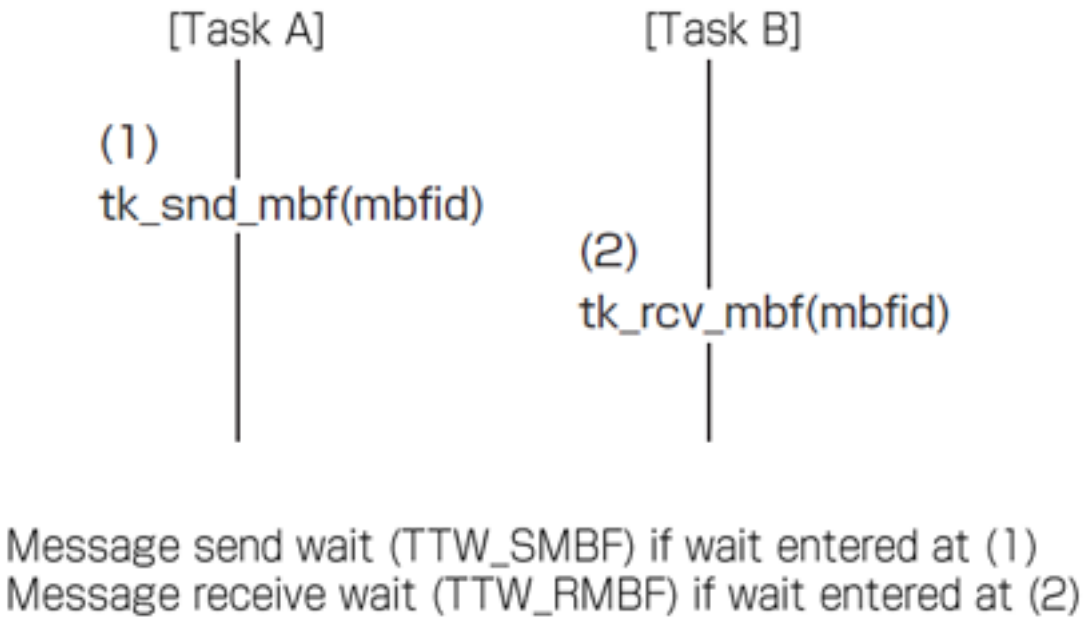

Figure 4.4: Synchronous Communication Using Message Buffer of bufsz = 0

4.5.2.2 tk_del_mbf - Delete Message Buffer

C Language Interface

```
#include <tk/tkernel.h>

ER ercd = tk_del_mbf (ID mbfid );
```

Parameter

ID	mbfid	Message Buffer ID	Message buffer ID

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 were tasks queued in the message buffer for message receipt or message sending, but error code E_DLT is returned to the tasks in WAITING state. 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.

4.5.2.3 tk_snd_mbf - Send Message to Message Buffer

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_snd_mbf (ID mbfid , CONST void *msg , INT msgsz , TMO tmout);

Parameter

ID	mbfid	Message Buffer ID	Message buffer ID
CONST void*	msg	Send Message	Start address of send message
INT	msgsz	Send Message Size	Send message size (in bytes)
TMO	tmout	Timeout	Timeout (ms)

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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, invalid msg, or tmout ≦ (-2))
E_DLT	The object being waited for was deleted (message buffer was deleted while waiting)
E_RLWAI	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion, or in dispatch disabled state)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO(* Available in some circumstances)

Description

tk_snd_mbf sends the message at the address specified in msg to the message buffer indicated in mbfid. The message size is specified in msgsz. This system call copies msgsz bytes starting from msg to the message queue of message buffer mbfid. The message queue is assumed to be implemented as a ring buffer.

If msgsz is larger than the maxmsz specified in 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 waiting state and is put in the send queue of the message buffer waiting for buffer space to become available. Waiting tasks are queued in either FIFO or priority order, depending on the attribute specified in 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 unit for tmout (time unit) is the same as that for system time (= 1 ms).

When TMO_POL = 0 is specified in tmout, it means 0 is specified as the timeout value, and if there is not enough buffer space, then E_TMOUT is returned without entering WAITING state. When TMO_FEVR (= -1) is specified 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, 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.

4.5.2.4 tk_snd_mbf_u - Send Message to Message Buffer (in microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_snd_mbf_u (ID mbfid , CONST void *msg , INT msgsz , TMO_U tmout_u);

Parameter

ID	mbfid	Message Buffer ID	Message buffer ID
CONST void*	msg	Send Message	Start address of send message
INT	msgsz	Send Message Size	Send message size (in bytes)
TMO_U	tmout_u	Timeout	Timeout (in microseconds)

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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, invalid msg, or tmout_u ≦ (-2))
E_DLT	The object being waited for was deleted (message buffer was deleted while waiting)
E_RLWAI	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion, or in dispatch disabled state)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO(* Available in certain circumstance)

Description

This system call takes 64-bit tmout_u in microseconds instead of the parameter tmout of tk_snd_mbf.

The specification of this system call is same as that of tk_snd_mbf, except that the parameter is replaced with tmout_u. For more details, see the description of tk_snd_mbf.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

4.5.2.5 tk_rcv_mbf - Receive Message from Message Buffer

C Language Interface

```
#include <tk/tkernel.h>
```

INT msgsz = tk_rcv_mbf (ID mbfid , void *msg , TMO tmout);

Parameter

ID	mbfid	Message Buffer ID	Message buffer ID
void*	msg	Receive Message	Address of the receive message
TMO	tmout	Timeout	Timeout (ms)

Return Parameter

INT	msgsz	Receive Message Size	Received message size (in bytes)
	or	Error Code	Error code

Error Code

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 (invalid msg, or tmout ≦ (-2))
E_DLT	The object being waited for was deleted (message buffer was deleted while waiting)
E_RLWAI	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion, or in dispatch disabled state)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

tk_rcv_mbf receives a message from the message buffer specified in mbfid, copying 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 WAITING 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 unit 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 WAITING 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.

4.5.2.6 tk_rcv_mbf_u - Receive Message from Message Buffer (in microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

INT msgsz = tk_rcv_mbf_u (ID mbfid , void *msg , TMO_U tmout_u);

Parameter

ID	`mbfid`	Message Buffer ID	Message buffer ID
void*	`msg`	Receive Message	Address of the receive message
TMO_U	`tmout_u`	Timeout	Timeout (in microseconds)

Return Parameter

INT	`msgsz`	Receive Message Size	Received message size (in bytes)
	or	Error Code	Error code

Error Code

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 (invalid `msg`, or `tmout_u` ≦ (-2))
E_DLT	The object being waited for was deleted (message buffer was deleted while waiting)
E_RLWAI	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion, or in dispatch disabled state)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

This system call takes 64-bit `tmout_u` in microseconds instead of the parameter `tmout` of tk_rcv_mbf.

The specification of this system call is same as that of tk_rcv_mbf, except that the parameter is replaced with `tmout_u`. For more details, see the description of tk_rcv_mbf.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

4.5.2.7 tk_ref_mbf - Reference Message Buffer Status

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_mbf (ID mbfid , T_RMBF *pk_rmbf);

Parameter

ID	mbfid	Message Buffer ID	Message buffer ID
T_RMBF*	pk_rmbf	Packet to Return Message Buffer Status	Pointer to the area to return the message buffer status

Return Parameter

ER	ercd	Error Code	Error code

pk_rmbf Detail:

void*	exinf	Extended Information	Extended information
ID	wtsk	Waiting Task ID	Receive waiting task ID
ID	stsk	Send Waiting Task ID	Send waiting task ID
INT	msgsz	Message Size	Size of the next message to be received (in bytes)
INT	frbufsz	Free Buffer Size	Free buffer size (in bytes)
INT	maxmsz	Maximum Message Size	Maximum message size (in bytes)

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

Error Code

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 (invalid pk_rmbf)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

References the status of the message buffer specified in mbfid, passing in the return parameters the send waiting task ID(stsk), the size of the next message to be received (msgsz), free buffer size (frbufsz), maximum message size (maxmsz), receive waiting task ID (wtsk), and extended information (exinf).

wtsk indicates the ID of a task waiting to receive a message from the message buffer. stsk indicates the ID of a task waiting to send a message to the message buffer. 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 which the message queue consists. This value indicates the approximate size of messages that can be sent.

The maximum message size as specified in tk_cre_mbf is returned to `maxmsz`.

4.5.3 Rendezvous

Rendezvous is a function to perform synchronized communication between tasks that are in a relationship of server and client. Specifically, rendezvous includes a function that enables both the client and server side tasks wait for the acceptance of processing, a function that enables the client side task send a message requesting a processing (call message) to the server side task, a function that enables the client side task wait for the completion of processing of server side task, and a function that enables the server side task reply a message of processing result (reply message) to the client side task. A series of processing steps listed above can be achieved easily by using system calls for rendezvous. Rendezvous works on the object that is called a rendezvous port.

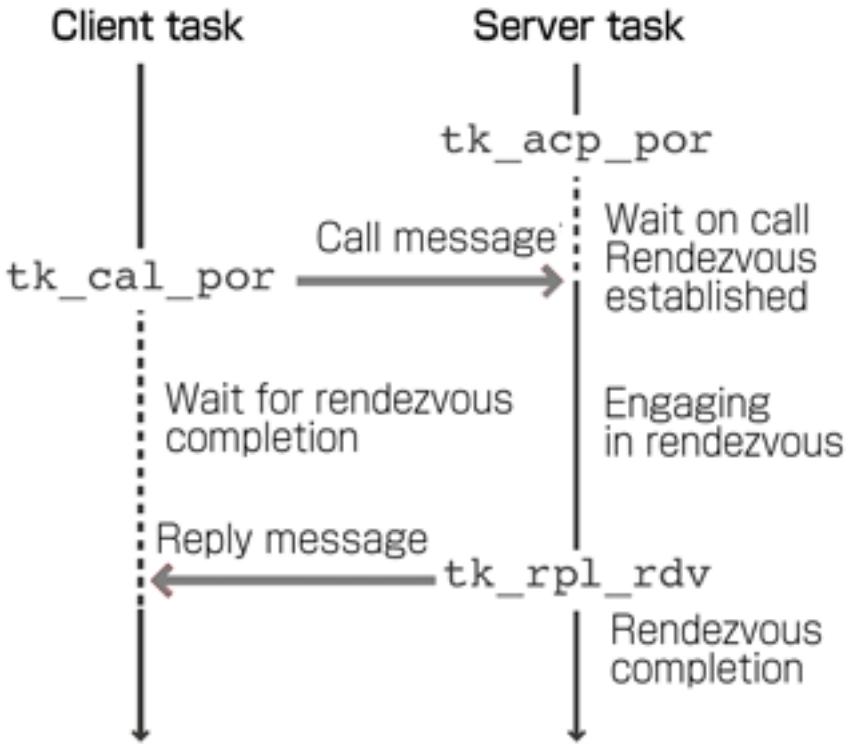

Figure 4.5: Rendezvous operation between a client task and server task

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. The ID number for the rendezvous port is called a rendezvous port ID.

A 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 condition bit pattern of the task calling a rendezvous for a rendezvous port and the rendezvous condition bit pattern of the accepting task) is not 0, the rendezvous is established. The state of the task calling the rendezvous is WAITING on rendezvous call until the rendezvous is established. The state of the task accepting a rendezvous is WAITING 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 WAITING for rendezvous completion until

the requested processing is completed. The task accepting the rendezvous is released from WAITING 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 WAITING state of the task that called the rendezvous is released.

The above operation is explained using the example shown in Figure 4.6, "Rendezvous Operation". In this example Task A and Task B run asynchronously.

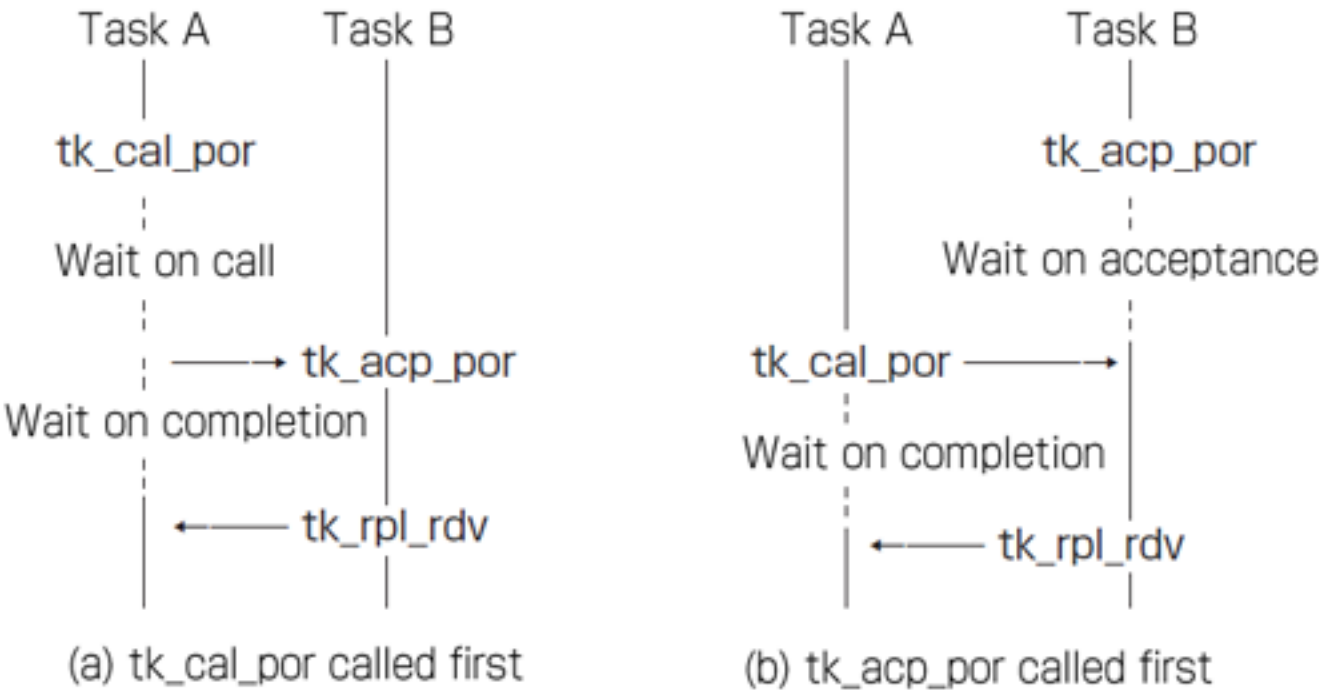

Figure 4.6: Rendezvous Operation

- If Task A first calls tk_cal_por, Task A goes to WAITING state until Task B calls tk_acp_por. The state of Task A at this time is WAITING on rendezvous call [Figure 4.6, "Rendezvous Operation" (a)].
- If, on the other hand, Task B first calls tk_acp_por, Task B goes to WAITING state until Task A calls tk_cal_por. The state of Task B at this time is WAITING on rendezvous acceptance [Figure 4.6, "Rendezvous Operation" (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 WAITING state while the WAITING state of Task B is released. The state of Task A is WAITING for rendezvous completion.
- The Task A WAITING state is released when Task B calls tk_rpl_rdv. Thereafter both tasks enter a run state.

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 a unique number called a rendezvous number to identify each rendezvous when more than one is established at the same time. 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. Even if the same task makes multiple rendezvous calls, the first rendezvous and second rendezvous must have different rendezvous numbers assigned.

Additional Notes
An example of the method to assign a rendezvous number is to use the ID number of the task that called a rendezvous to the lower bits of the rendezvous number, and put a serial number to the higher bits.

Rationale for the Specification
The name "rendezvous" of this function is based on the fact that a client side task and a server side task have a rendezvous between them. When rendezvous was included in T-Kernel specification, Rendezvous in Ada programming language and CSP (Communicating Sequential Processes) from which Ada derived affected it. However, the rendezvous function provided by T-Kernel is not the same as that of Ada language.
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 WAITING state for its completion, will have its WAITING 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 WAITING state for rendezvous completion is made to remember the unique number of the rendezvous for which it is waiting, error will be returned when the attempt is made to terminate the first rendezvous.

4.5.3.1 tk_cre_por - Create Port for Rendezvous

C Language Interface

```
#include <tk/tkernel.h>
```

ID porid = tk_cre_por (CONST T_CPOR *pk_cpor);

Parameter

CONST T_CPOR*	pk_cpor	Packet to Create Port	Rendezvous port creation information

pk_cpor Detail:

void*	exinf	Extended Information	Extended information
ATR	poratr	Port Attribute	Rendezvous port attributes
INT	maxcmsz	Max Call Message Size	Maximum call message size (in bytes)
INT	maxrmsz	Max Reply Message Size	Maximum reply message size (in bytes)
UB	dsname[8]	DS Object name	DS object name

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

Return Parameter

ID	porid	Port ID	Rendezvous port ID
	or	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Creates a rendezvous port, assigning to it a rendezvous port ID number. This specification allocates a control block to the created rendezvous port. A rendezvous port is an object used as an OS primitive for implementing a rendezvous capability.

exinf can be used freely by the user to set 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 message buffer 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.

poratr indicates system attributes in its lower bits and implementation-dependent attributes in its higher bits. The system attribute part of poratr is specified as follows.

```
poratr:= (TA_TFIFO || TA_TPRI) | [TA_DSNAME] | [TA_NODISWAI]
```

TA_TFIFO	Tasks waiting on call are queued in FIFO order
TA_TPRI	Tasks waiting on call are queued in priority order
TA_DSNAME	Specifies DS object name
TA_NODISWAI	Disabling of wait by tk_dis_wai is prohibited

TA_TFIFO and TA_TPRI specify the queuing order of tasks waiting on a rendezvous call. Tasks waiting on rendezvous acceptance are queued in FIFO order only.

When TA_DSNAME is specified, dsname is valid and specifies the DS object name. DS object name is used to identify objects by debugger, and it is handled only by T-Kernel/DS API, td_ref_dsname and td_set_dsname. For more details, see the description of td_ref_dsname and td_set_dsname. If TA_DSNAME is not specified, dsname is ignored. Then td_ref_dsname and td_set_dsname return E_OBJ error.

```
#define TA_TFIFO        0x00000000      /* manage queue by FIFO */
#define TA_TPRI         0x00000001      /* manage queue by priority */
#define TA_DSNAME       0x00000040      /* DS object name */
#define TA_NODISWAI     0x00000080      /* reject request to disable wait */
```

maxcmsz specifies the maximum size (bytes) of the message passed at rendezvous call. maxcmsz can be 0. When maxcmsz is 0, the size of the message passed at rendezvous calling is limited to 0, and thus it is used only for synchronization without message.

maxrmsz specifies the maximum size (bytes) of the message passed at rendezvous return. maxrmsz can be 0. When maxrmsz is 0, the size of the message passed at rendezvous return is limited to 0.

4.5.3.2 tk_del_por - Delete Port for Rendezvous

C Language Interface

```
#include <tk/tkernel.h>

ER ercd = tk_del_por (ID porid );
```

Parameter

ID	porid	Port ID	Rendezvous port ID

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 WAITING 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 WAITING state), and the state of the task calling the rendezvous (WAITING for rendezvous completion) remains unchanged. When the task accepting the rendezvous issues tk_rpl_rdv, that tk_rpl_rdv will execute normally even if the port on which the rendezvous was established has been deleted.

4.5.3.3 tk_cal_por - Call Port for Rendezvous

C Language Interface

```
#include <tk/tkernel.h>
```

INT rmsgsz = tk_cal_por (ID porid , UINT calptn , void *msg , INT cmsgsz , TMO tmout);

Parameter

ID	porid	Port ID	Rendezvous port ID
UINT	calptn	Call Bit Pattern	Call bit pattern (indicating conditions of the caller)
void*	msg	Message	Address of the message
INT	cmsgsz	Call Message Size	Call message size (bytes)
TMO	tmout	Timeout	Timeout (ms)

Return Parameter

INT	rmsgsz	Reply Message Size	Reply message size (in bytes)
	or	Error Code	Error code

Error Code

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, invalid msg, or tmout ≦ (-2))
E_DLT	The object being waited for was deleted (the rendezvous port was deleted while waiting)
E_RLWAI	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion, or in dispatch disabled state)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 tk_cal_por. 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 WAITING 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 WAITING state on rendezvous call. The order of tasks in the call queue is either FIFO or priority order, depending on the attribute made 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` 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 tk_cal_por return parameter. The original content of the message area passed in `msg` by tk_cal_por ends up being overwritten by the reply message received when tk_rpl_rdv executes.

Note that it is possible message content 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 made before a task enters WAITING state on rendezvous call; and if error is detected, the task executing tk_cal_por does not enter WAITING state.

A maximum wait time (timeout) until rendezvous establishment 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 unit 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 WAITING 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.

`tmout` indicates the time allowed for a rendezvous to be established, and does not apply to the time from rendezvous establishment to rendezvous completion.

4.5.3.4 tk_cal_por_u - Call Port for Rendezvous (in microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

INT rmsgsz = tk_cal_por_u (ID porid , UINT calptn , void *msg , INT cmsgsz , TMO_U tmout_u);

Parameter

ID	porid	Port ID	Rendezvous port ID
UINT	calptn	Call Bit Pattern	Call bit pattern (indicating conditions of the caller)
void*	msg	Message	Address of the message
INT	cmsgsz	Call Message Size	Call message size (bytes)
TMO_U	tmout_u	Timeout	Timeout (in microseconds)

Return Parameter

INT	rmsgsz	Reply Message Size	Reply message size (in bytes)
	or	Error Code	Error code

Error Code

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, invalid msg, or tmout_u ≦ (-2))
E_DLT	The object being waited for was deleted (the rendezvous port was deleted while waiting)
E_RLWAI	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion, or in dispatch disabled state)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

This system call takes 64-bit tmout_u in microseconds instead of the parameter tmout of tk_cal_por.

The specification of this system call is same as that of tk_cal_por, except that the parameter is replaced with tmout_u. For more details, see the description of tk_cal_por.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

4.5.3.5 tk_acp_por - Accept Port for Rendezvous

C Language Interface

```
#include <tk/tkernel.h>
```

INT cmsgsz = tk_acp_por (ID porid , UINT acpptn , RNO *p_rdvno , void *msg , TMO tmout);

Parameter

ID	porid	Port ID	Rendezvous port ID
UINT	acpptn	Accept Bit Pattern	Accept bit pattern (indicating conditions for acceptance)
RNO*	p_rdvno	Pointer to Rendezvous Number	Pointer to the area to return the return parameter rdvno
void*	msg	Packet of Call Message	Address of call message packet
TMO	tmout	Timeout	Timeout (ms)

Return Parameter

RNO	rdvno	Rendezvous Number	Rendezvous number
INT	cmsgsz	Call Message Size	Call message size (bytes)
		or Error Code	Error code

Error Code

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 (acpptn = 0, invalid msg, or tmout ≦ (-2))
E_DLT	The object being waited for was deleted (the rendezvous port was deleted while waiting)
E_RLWAI	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion, or in dispatch disabled state)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 WAITING state on

rendezvous acceptance for that port. No error results if there is already another task in WAITING 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 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 sides.

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 return value 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 information 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 side 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 unit 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 WAITING 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.

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 as much as 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 WAITING 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 WAITING 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 lower byte of `rdvno`, using the higher byte for a serial 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 approach equivalent to an Ada select statement sample (Figure 4.7, "Sample Ada-like Program Using select Statement") is shown in Figure 4.8, "Using Rendezvous to Implement Ada select Function".

```
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 4.7: Sample Ada-like Program Using select Statement

- 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 will look like the following:

```
ptn := 0;
if condition_A then ptn := ptn + 2^0 endif;
if condition_B then ptn := ptn + 2^1 endif;
if condition_C then ptn := ptn + 2^2 endif;
tk_acp_por(acpptn := ptn);
```

- If the program contains a simple entry_A accept with no select in addition to the select statement shown above,

```
tk_acp_por(acpptn := 2^0);
```

  can be executed. If it is desired to have entry_A, entry_B, and entry_C wait unconditionally in parallel (using OR)

```
tk_acp_por(acpptn := 2^2+2^1+2^0);
```

  can be executed.

- If the caller can call entry_A by the following

```
tk_cal_por(calptn := 2^0);
```

  and if the call is for entry_C,

```
tk_cal_por(calptn := 2^2);
```

  can be executed.

Figure 4.8: Using Rendezvous to Implement Ada select Function

The Ada select function is provided only on the accepting side, but it is also possible to implement a select function on the calling side by specifying multiple bits in `calptn` .

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 each other on the calling and accepting sides is because processing required after a rendezvous is established differs for the tasks on each side. That is, whereas the calling task enters WAITING state after the rendezvous is established, the accepting task enters READY state.

4.5.3.6 tk_acp_por_u - Accept Port for Rendezvous (in microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

INT cmsgsz = tk_acp_por_u (ID porid , UINT acpptn , RNO *p_rdvno , void *msg , TMO_U tmout_u);

Parameter

ID	porid	Port ID	Rendezvous port ID
UINT	acpptn	Accept Bit Pattern	Accept bit pattern (indicating conditions for acceptance)
RNO*	p_rdvno	Pointer to Rendezvous Number	Pointer to the area to return the return parameter rdvno
void*	msg	Packet of Call Message	Address of call message packet
TMO_U	tmout_u	Timeout	Timeout (in microseconds)

Return Parameter

RNO	rdvno	Rendezvous Number	Rendezvous number
INT	cmsgsz	Call Message Size	Call message size (bytes)
		or Error Code	Error code

Error Code

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 (acpptn = 0, invalid msg, or tmout_u ≦ (-2))
E_DLT	The object being waited for was deleted (the rendezvous port was deleted while waiting)
E_RLWAI	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion, or in dispatch disabled state)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

This system call takes 64-bit tmout_u in microseconds instead of the parameter tmout of tk_acp_por.

The specification of this system call is same as that of tk_acp_por, except that the parameter is replaced with tmout_u. For more details, see the description of tk_acp_por.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

4.5.3.7 tk_fwd_por - Forwards rendezvous to other port

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_fwd_por (ID porid , UINT calptn , RNO rdvno , CONST void *msg , INT cmsgsz);

Parameter

ID	porid	Port ID	Destination rendezvous port ID
UINT	calptn	Call Bit Pattern	Call bit pattern (indicating conditions of the caller)
RNO	rdvno	Rendezvous Number	Rendezvous number before transmission
CONST void*	msg	Call Message	Address of forwarded message packet
INT	cmsgsz	Call Message Size	Forwarded message size (in bytes)

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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 (cmsgsz < 0, cmsgsz > maxcmsz after forwarding, cmsgsz > maxrmsz before forwarding, calptn = 0, or invalid msg)
E_OBJ	Invalid object state (invalid rdvno, or maxrmsz after forwarding > maxrmsz before forwarding)
E_CTX	Context error (issued from task-independent portion (implementation-dependent error))
E_DISWAI	Wait released due to disabling of wait

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 WAITING 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 specified in `calptn` by Task Y when it called tk_cal_por, but those 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).

As the result, the kernel does not have to remember the history of rendezvous forwarding.

If tk_ref_tsk is executed for a task that has returned to WAITING 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 the WAITING 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 by tk_fwd_por is copied to another memory area (such as the message area specified by 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 prepares 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 Notes 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 tk_fwd_por or tk_rpl_rdv 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 made for issuing of tk_fwd_por or tk_rpl_rdv from a task-independent portion is implementation-dependent.

When as a result of tk_fwd_por Task Y that was in WAITING state for rendezvous completion reverts to WAITING 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.

Additional Notes

A server task operation using tk_fwd_por is illustrated in Figure 4.9, “Server Task Operation Using tk_fwd_por”.[2]

[2]

- 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.

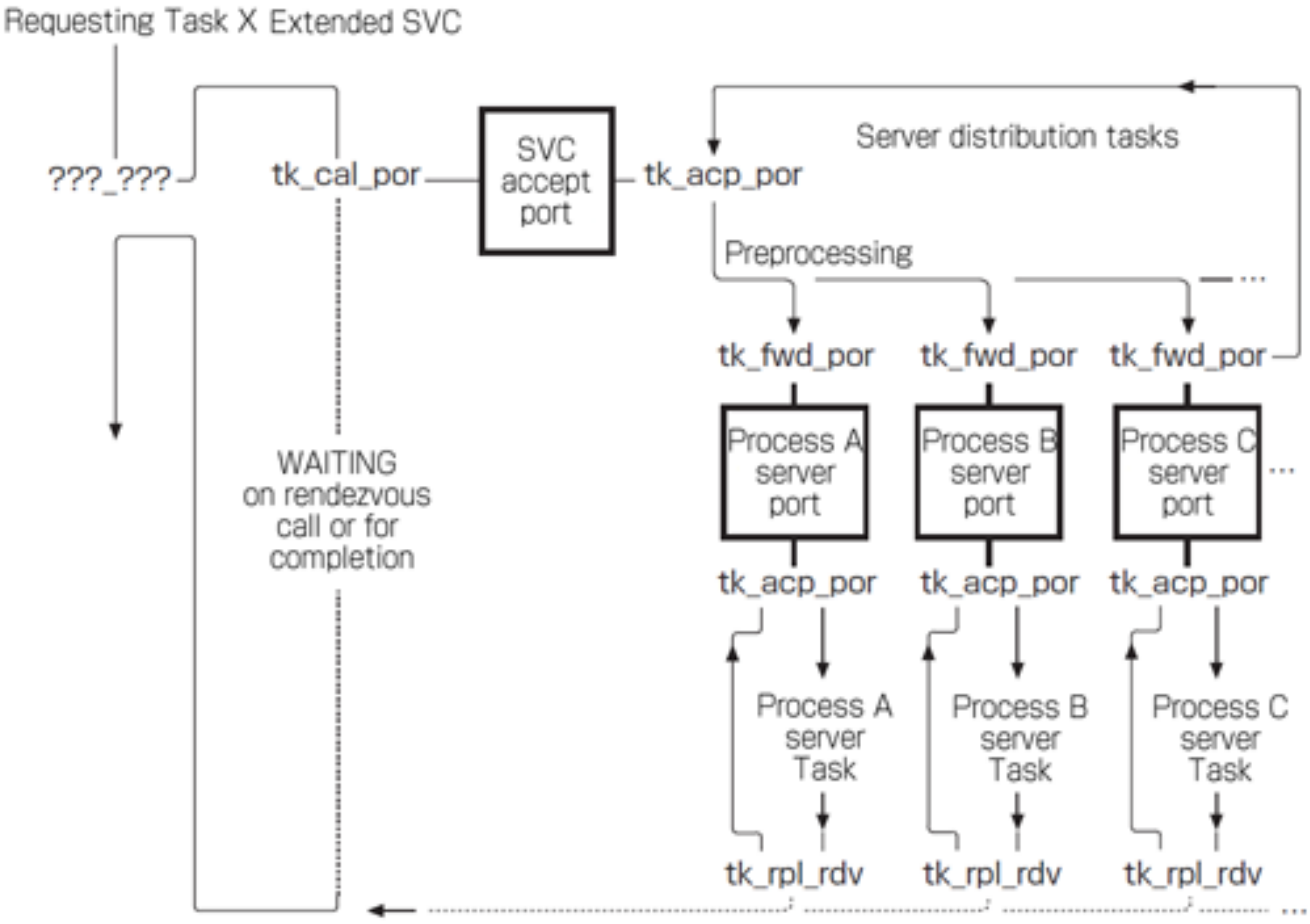

Figure 4.9: Server Task Operation Using tk_fwd_por

Generally tk_fwd_por is executed by server distribution tasks (tasks for distributing server-accepted processing to other tasks) as shown in Figure 4.9, "Server Task Operation Using tk_fwd_por". Accordingly, a server distribution task that has executed tk_fwd_por must go on to accept 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, an implementation 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 is specified to allow this sort of implementation.

- 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 expected reply message size.
- 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 does not become larger than `maxrmsz` of the port before forwarding. The former is equal to or smaller than the latter.

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.

4.5.3.8 tk_rpl_rdv - Reply Rendezvous

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_rpl_rdv (RNO rdvno , CONST void *msg , INT rmsgsz);

Parameter

RNO	rdvno	Rendezvous Number	Rendezvous number
CONST void*	msg	Reply Message	Address of the reply message
INT	rmsgsz	Reply Message Size	Reply message size (in bytes)

Return Parameter

ER	ercd	Error Code	Error code

Error Code

E_OK	Normal completion
E_PAR	Parameter error (rmsgsz < 0, rmsgsz > maxrmsz, or invalid msg)
E_OBJ	Invalid object state (rdvno is invalid)
E_CTX	Context error (issued from task-independent portion (implementation-dependent error))

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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; that is, 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 Task Y state goes from WAITING 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 set in 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 WAITING state for rendezvous completion.

It is not possible to issue tk_fwd_por and tk_rpl_rdv from a task-independent portion, but it is possible to issue tk_fwd_por or tk_rpl_rdv 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 made for issuing of tk_fwd_por or tk_rpl_rdv 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 WAITING state, or in another area that can be referenced from the TCB, such as a stack area.

Rationale for the Specification

The parameter `rdvno` is passed to tk_rpl_rdv and tk_fwd_por as information for distinguishing a 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.

4.5.3.9 tk_ref_por - Reference Port Status

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_por (ID porid , T_RPOR *pk_rpor);

Parameter

ID	porid	Port ID	Rendezvous port ID
T_RPOR*	pk_rpor	Packet to Return Port Status	Pointer to the area to return the rendezvous port status

Return Parameter

ER	ercd	Error Code	Error code

pk_rpor Detail:

void*	exinf	Extended Information	Extended information
ID	wtsk	Waiting Task ID	Call waiting task ID
ID	atsk	Accept Waiting Task ID	Accept waiting task ID
INT	maxcmsz	Maximum Call Message Size	Maximum call message size (in bytes)
INT	maxrmsz	Maximum Reply Message Size	Maximum reply message size (in bytes)

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

Error Code

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 (invalid pk_rpor)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

References the status of the rendezvous port specified in porid, passing in return parameters the accept waiting task ID (atsk), the call waiting task ID (wtsk), the maximum message sizes (maxcmsz, maxrmsz), and the extended information (exinf).

wtsk indicates the ID of a task in WAITING 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 WAITING 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.

Additional Notes

This system call cannot be used to get information about tasks involved in a currently established rendezvous.

4.6 Memory Pool Management Functions

Memory pool management functions are for managing memory pools and allocating memory blocks by using software.

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. The ID number for the fixed-size memory pool is 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 WAITING 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.

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.

4.6.1.1 tk_cre_mpf - Create Fixed-size Memory Pool

C Language Interface

```
#include <tk/tkernel.h>
```

ID mpfid = tk_cre_mpf (CONST T_CMPF *pk_cmpf);

Parameter

CONST T_CMPF*	pk_cmpf	Packet to Create Memory Pool	Information about the fixed-size memory pool to be created

pk_cmpf Detail:

void*	exinf	Extended Information	Extended information
ATR	mpfatr	Memory Pool Attribute	Memory pool attribute
INT	mpfcnt	Memory Pool Block Count	Memory pool block count
INT	blfsz	Memory Block Size	Fixed-size memory block size (in bytes)
UB	dsname[8]	DS Object name	DS object name

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

Return Parameter

ID	mpfid		Memory Pool ID	Fixed-size memory pool ID
		or	Error Code	Error code

Error Code

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, or mpfcnt or blfsz is negative or invalid)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Creates a fixed-size memory pool, assigning to it 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 set 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 may need 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 kernel pays no attention to the contents of exinf.

mpfatr indicates system attributes in its lower bits and implementation-dependent attributes in its higher bits. The system attribute part of mpfatr is as follows.

```
mbxatr:= (TA_TFIFO || TA_TPRI) | [TA_DSNAME] | [TA_NODISWAI]
       | (TA_RNG0 || TA_RNG1 || TA_RNG2 || TA_RNG3)
```

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_DSNAME	Specifies DS object name
TA_NODISWAI	Disabling of wait by tk_dis_wai is prohibited

```
#define TA_TFIFO        0x00000000      /* manage queue by FIFO */
#define TA_TPRI         0x00000001      /* manage queue by priority */
#define TA_DSNAME       0x00000040      /* DS object name */
#define TA_NODISWAI     0x00000080      /* reject request to disable wait */
#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 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 from 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 an 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_DSNAME is specified, dsname is valid and specifies the DS object name. DS object name is used to identify objects by debugger, and it is handled only by T-Kernel/DS API, td_ref_dsname and td_set_dsname. For more details, see the description of td_ref_dsname and td_set_dsname. If TA_DSNAME is not specified, dsname is ignored. Then td_ref_dsname and td_set_dsname return E_OBJ error.

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 handle all TA_RNGn as equivalent to TA_RNG0, but error must not be returned.

4.6.1.2 tk_del_mpf - Delete Fixed-size Memory Pool

C Language Interface

```
#include <tk/tkernel.h>

ER ercd = tk_del_mpf (ID mpfid );
```

Parameter

ID	mpfid	Memory Pool ID	Fixed-size memory pool ID

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 not all blocks have 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 WAITING state.

4.6.1.3 tk_get_mpf - Get Fixed-size Memory Block

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_get_mpf (ID mpfid , void **p_blf , TMO tmout);

Parameter

ID	`mpfid`	Memory Pool ID	Fixed-size memory pool ID
void**	`p_blf`	Pointer to Block Start Address	Pointer to the area to return the block start address `blf`
TMO	`tmout`	Timeout	Timeout (ms)

Return Parameter

ER	`ercd`	Error Code	Error code
void*	`blf`	Block Start Address	Memory block start address

Error Code

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	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion, or in dispatch disabled state)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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_mpf 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 unit 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 WAITInG 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.

4.6.1.4 tk_get_mpf_u - Get Fixed-size Memory Block (Microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_get_mpf_u (ID mpfid , void **p_blf , TMO_U tmout_u);

Parameter

ID	`mpfid`	Memory Pool ID	Fixed-size memory pool ID
void**	`p_blf`	Pointer to Block Start Address	Pointer to the area to return the block start address `blf`
TMO_U	`tmout_u`	Timeout	Timeout (in microseconds)

Return Parameter

ER	`ercd`	Error Code	Error code
void*	`blf`	Block Start Address	Memory block start address

Error Code

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_u` ≦ (-2))
E_DLT	The object being waited for was deleted (the memory pool was deleted while waiting)
E_RLWAI	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion, or in dispatch disabled state)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

This system call takes 64-bit `tmout_u` in microseconds instead of the parameter `tmout` of tk_get_mpf.

The specification of this system call is same as that of tk_get_mpf, except that the parameter is replaced with `tmout_u`. For more details, see the description of tk_get_mpf.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

4.6.1.5 tk_rel_mpf - Release Fixed-size Memory Block

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_rel_mpf (ID mpfid , void *blf);

Parameter

ID	mpfid	Memory Pool ID	Fixed-size memory pool ID
void*	blf	Block Start Address	Memory block start address

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 WAITING 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 performed or not is implementation-dependent.

4.6.1.6 tk_ref_mpf - Reference Fixed-size Memory Pool Status

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_mpf (ID mpfid , T_RMPF *pk_rmpf);

Parameter

ID	mpfid	Memory Pool ID	Fixed-size memory pool ID
T_RMPF*	pk_rmpf	Packet to Return Memory Pool Status	Pointer to the area to return the memory pool status

Return Parameter

ER	ercd	Error Code	Error code

pk_rmpf Detail:

void*	exinf	Extended Information	Extended information
ID	wtsk	Waiting Task ID	Waiting task ID
INT	frbcnt	Free Block Count	Free block count

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

Error Code

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 (invalid pk_rmpf)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 ID (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.

Additional Notes

Whereas `frsz` returned by tk_ref_mpl gives the total free memory size in bytes, `frbcnt` returned by tk_ref_mpf gives the number of unused memory blocks.

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. The ID number for the variable-size memory pool is 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 WAITING 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.

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-byte block of space are free, Task B is made to wait until Task A has acquired the requested memory block.

4.6.2.1 tk_cre_mpl - Create Variable-size Memory Pool

C Language Interface

```
#include <tk/tkernel.h>
```

ID mplid = tk_cre_mpl (CONST T_CMPL *pk_cmpl);

Parameter

CONST T_CMPL*	pk_cmpl	Packet to Create Memory Pool	Information about the variable-size memory pool to be created

pk_cmpl Detail:

void*	exinf	Extended Information	Extended information
ATR	mplatr	Memory Pool Attribute	Memory pool attribute
INT	mplsz	Memory Pool Size	Memory pool size (in bytes)
UB	dsname[8]	DS Object name	DS object name

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

Return Parameter

ID	mplid	Memory Pool ID	Variable-size memory pool ID
	or	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Creates a variable-size memory pool, assigning to it 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 assigns a control block to the memory pool.

exinf can be used freely by the user to set 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 may need 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 kernel pays no attention to the contents of exinf.

mplatr indicates system attributes in its lower bits and implementation-dependent attributes in its higher bits. The system attribute part of mplatr is as follows.

```
mplatr:= (TA_TFIFO || TA_TPRI) | [TA_DSNAME] | [TA_NODISWAI]
       | (TA_RNG0 || TA_RNG1 || TA_RNG2 || TA_RNG3)
```

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_DSNAME	Specifies DS object name
TA_NODISWAI	Disabling of wait by tk_dis_wai is prohibited

```
#define TA_TFIFO        0x00000000      /* manage task queue by FIFO */
#define TA_TPRI         0x00000001      /* manage task queue by priority */
#define TA_DSNAME       0x00000040      /* DS object name */
#define TA_NODISWAI     0x00000080      /* reject request to disable wait */
#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 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.

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-byte block of space are freed by tk_rel_mpl of another task, Task B is made to wait until Task A has acquired the requested memory block.

TA_RNGn is specified to limit the protection levels from 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 an 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_DSNAME is specified, dsname is valid and specifies the DS object name. DS object name is used to identify objects by debugger, and it is handled only by T-Kernel/DS API, td_ref_dsname and td_set_dsname. For more details, see the description of td_ref_dsname and td_set_dsname. If TA_DSNAME is not specified, dsname is ignored. Then td_ref_dsname and td_set_dsname return E_OBJ error.

Additional Notes

If the task at the head of the queue waiting for memory allocation has its WAITING state forcibly released, or if a different task becomes the first in the queue as a result of a change in task priority, memory allocation is attempted to that task. If memory can be allocated, the WAITInG state of that task is released. In this way it is possible under some circumstances for memory allocation to take place and task WAITING 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 handle all TA_RNGn as equivalent to TA_RNG0, but error must not be returned.

Rationale for the Specification

The capability of creating multiple variable-size memory pools can be used for memory allocation as needed for error handling or in emergent situations in programming, etc.

4.6.2.2 tk_del_mpl - Delete Variable-size Memory Pool

C Language Interface

```
#include <tk/tkernel.h>

ER ercd = tk_del_mpl (ID mplid );
```

Parameter

ID	mplid	Memory Pool ID	Variable-size memory pool ID

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 not all blocks have 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 WAITING state.

4.6.2.3 tk_get_mpl - Get Variable-size Memory Block

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_get_mpl (ID mplid , INT blksz , void **p_blk , TMO tmout);

Parameter

ID	`mplid`	Memory Pool ID	Variable-size memory pool ID
INT	`blksz`	Memory Block Size	Memory block size (in bytes)
void**	`p_blk`	Pointer to Block Start Address	Pointer to the area to return the block start address `blk`
TMO	`tmout`	Timeout	Timeout (ms)

Return Parameter

ER	`ercd`	Error Code	Error code
void*	`blk`	Block Start Address	Memory block start address

Error Code

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	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion, or in dispatch disabled state)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 WAITING 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 unit 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 WAITING 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.

4.6.2.4 tk_get_mpl_u - Get Variable-size Memory Block (Microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_get_mpl_u (ID mplid , INT blksz , void **p_blk , TMO_U tmout_u);

Parameter

ID	`mplid`	Memory Pool ID	Variable-size memory pool ID
INT	`blksz`	Memory Block Size	Memory block size (in bytes)
void**	`p_blk`	Pointer to Block Start Address	Pointer to the area to return the block start address `blk`
TMO_U	`tmout_u`	Timeout	Timeout (in microseconds)

Return Parameter

ER	`ercd`	Error Code	Error code
void*	`blk`	Block Start Address	Memory block start address

Error Code

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_u` ≦ (-2))
E_DLT	The object being waited for was deleted (the memory pool was deleted while waiting)
E_RLWAI	Waiting state released (tk_rel_wai received in waiting state)
E_DISWAI	Wait released due to disabling of wait
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion, or in dispatch disabled state)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

This system call takes 64-bit `tmout_u` in microseconds instead of the parameter `tmout` of tk_get_mpl.

The specification of this system call is same as that of tk_get_mpl, except that the parameter is replaced with `tmout_u`. For more details, see the description of tk_get_mpl.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

4.6.2.5 tk_rel_mpl - Release Variable-size Memory Block

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_rel_mpl (ID mplid , void *blk);

Parameter

ID	mplid	Memory Pool ID	Variable-size memory pool ID
void*	blk	Block Start Address	Memory block start address

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 WAITING 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 this error detection is performed or not is implementation-dependent.

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 their requested memory size. The task precedence among tasks of the same priority after their WAITING state is released in such a case is the order in which they were queued.

4.6.2.6 tk_ref_mpl - Reference Variable-size Memory Pool Status

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_mpl (ID mplid , T_RMPL *pk_rmpl);

Parameter

ID	mplid	Memory Pool ID	Variable-size memory pool ID
T_RMPL*	pk_rmpl	Packet to Return Memory Pool Status	Pointer to the area to return the memory pool status

Return Parameter

ER	ercd	Error Code	Error code

pk_rmpl Detail:

void*	exinf	Extended Information	Extended information
ID	wtsk	Waiting Task ID	Waiting task ID
INT	frsz	Free Memory Size	Free memory size (in bytes)
INT	maxsz	Max Memory Size	Maximum memory space size (in bytes)

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

Error Code

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 (invalid pk_rmpl)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

References the status of the variable-size memory pool specified in mplid, passing in return parameters the total size of free space (frsz), the maximum size of memory immediately available (maxsz), the waiting task ID (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.

4.7 Time Management Functions

Time management functions perform time-dependent processing. They include functions for system time management, cyclic handlers, and alarm handlers.

The generic name used in the following for cyclic handlers and alarm handlers is time event handlers.

4.7.1 System Time Management

System time management functions manipulate system time. Functions are provided for system clock setting and reference, and for referencing system operating time.

4.7.1.1 tk_set_tim - Set Time

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_set_tim (CONST SYSTIM *pk_tim);

Parameter

CONST SYSTIM*	pk_tim	Packet of Current Time	Packet indicating current time (ms)

pk_tim Detail:

W	hi	High 32 bits	Higher 32 bits of current time for setting the system time
UW	lo	Low 32 bits	Lower 32 bits of current time for setting the system time

Return Parameter

ER	ercd	Error Code	Error code

Error Codes

E_OK	Normal completion
E_PAR	Parameter error (pk_tim is invalid, or time setting is invalid)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Sets the system clock to the value specified in pk_tim.

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.

The time specified in pk_tim for tk_set_tim() is not restricted to the resolution of the timer interrupt cycle. But the time that is read later by tk_get_tim() changes according to the time resolution of the timer interrupt cycle. For example, in the system where the timer interrupt cycle is 10 milliseconds, if the time of 0005 (ms)

is specified in tk_set_tim(), then the time obtained later by tk_get_tim() changes as follows: 0005 (ms) → 0015 (ms) → 0025 (ms).

4.7.1.2 tk_set_tim_u - Set Time (in microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_set_tim_u (SYSTIM_U tim_u);

Parameter

SYSTIM_U	tim_u	Current Time	Current time (in microseconds)

Return Parameter

ER	ercd	Error Code	Error code

Error Codes

E_OK	Normal completion
E_PAR	Parameter error (tim_u is invalid, or time setting is invalid)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

This system call takes 64-bit tim_u in microseconds instead of the parameter pk_tim of tk_set_tim.

Whereas the parameter pk_tim of tk_set_tim is passed in packet using the structure SYSTIM, the parameter tim_u of tk_set_tim_u is passed by value (not packet) using the 64-bit signed integer SYSTIM_U.

The specification of this system call is same as that of tk_set_tim, except the above-mentioned point. For more details, see the description of tk_set_tim.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

4.7.1.3 tk_get_tim - Get System Time

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_get_tim (SYSTIM *pk_tim);

Parameter

SYSTIM*	pk_tim	Packet of Current Time	Pointer to the area to return the current time (ms)

Return Parameter

ER	ercd	Error Code	Error code

pk_tim Detail:

W	hi	High 32 bits	Higher 32 bits of current time of the system time
UW	lo	Low 32 bits	Lower 32 bits of current time of the system time

Error Codes

E_OK	Normal completion
E_PAR	Parameter error (pk_tim is invalid)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Reads the current value of the system clock and returns in it pk_tim.

System time is expressed as cumulative milliseconds from 0:00:00 (GMT), January 1, 1985.

The resolution of the current system time read by this system call varies depending on the time resolution of the timer interrupt interval (cycle).

Additional Notes

tk_get_tim() cannot be used to get the elapsed time that is shorter than the timer interrupt interval (cycle). To find out the elapsed time shorter than the timer interrupt interval (cycle), use the return parameter ofs of tk_get_tim_u() or td_get_tim().

4.7.1.4 tk_get_tim_u - Get System Time (Microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_get_tim_u (SYSTIM_U *tim_u , UINT *ofs);

Parameter

SYSTIM_U*	tim_u	Time	Pointer to the area to return the current time (microseconds)
UINT*	ofs	Offset	Pointer to the area to return the return parameter ofs

Return Parameter

ER	ercd	Error Code	Error code
SYSTIM_U	tim_u	Time	Current time (in microseconds)
UINT	ofs	Offset	Relative elapsed time from tim_u (nanoseconds)

Error Codes

E_OK	Normal completion
E_PAR	Parameter error (invalid tim_u or ofs)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

This system call takes 64-bit tim_u in microseconds instead of the return parameter pk_tim of tk_get_tim. It also includes the return parameter ofs that returns the relative time in nanoseconds.

tim_u has the resolution of time interrupt interval (cycle), but even more precise time information is obtained in ofs as the elapsed time from tim_u in nanoseconds. The resolution of ofs is implementation-dependent, but generally is the resolution of hardware timer.

If ofs = NULL is set, the information of ofs is not stored.

The specification of this system call is same as that of tk_get_tim, except the above-mentioned point. In addition, the specification of this system call is the same as that of td_get_tim, except that the data type of tim_u is SYSTIM_U. For more details, see the description of tk_get_tim and td_get_tim.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

4.7.1.5 tk_get_otm - Get Operating Time

C Language Interface

```
#include <tk/tkernel.h>

ER ercd = tk_get_otm (SYSTIM *pk_tim );
```

Parameter

SYSTIM*	pk_tim	Packet of Operating Time	Pointer to the area to return the operating time (ms)

Return Parameter

ER	ercd	Error Code	Error code

pk_tim Detail:

W	hi	High 32 bits	Higher 32 bits of the system operating time
UW	lo	Low 32 bits	Lower 32 bits of the system operating time

Error Codes

E_OK	Normal completion
E_PAR	Parameter error (pk_tim is invalid)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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.

4.7.1.6 tk_get_otm_u - Get Operating Time (Microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_get_otm_u (SYSTIM_U *tim_u , UINT *ofs);

Parameter

SYSTIM_U*	tim_u	Time	Pointer to the area to return the operating time (microseconds)
UINT*	ofs	Offset	Pointer to the area to return the return parameter ofs

Return Parameter

ER	ercd	Error Code	Error Codes
SYSTIM_U	tim_u	Time	Operating time (microseconds)
UINT	ofs	Offset	Relative elapsed time from tim_u (nanoseconds)

Error Codes

E_OK	Normal completion
E_PAR	Parameter error (invalid tim_u or ofs)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

This system call takes 64-bit tim_u in microseconds instead of the return parameter pk_tim of tk_get_otm. It also includes the return parameter ofs that returns the relative time in nanoseconds.

tim_u has the resolution of time interrupt interval (cycle), but even more precise time information is obtained in ofs as the elapsed time from tim_u in nanoseconds. The resolution of ofs is implementation-dependent, but generally is the resolution of hardware timer.

If ofs = NULL is set, the information of ofs is not stored.

The specification of this system call is same as that of tk_get_otm, except the above-mentioned point. In addition, the specification of this system call is the same as that of td_get_otm, except that the data type of tim_u is SYSTIM_U. For more details, see the description of tk_get_otm and td_get_otm.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

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. The ID number for the cyclic handler is 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 next be started 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 is implementation-dependent when the cycle phase is made longer than the cycle time.

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 a 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.

Additional Notes

Actual time resolution in T-Kernel time management functions processing uses one that is specified by the "timer interrupt interval" (`TTimPeriod`) in Section 5.7.2, “Standard System Configuration Information”. It also means that a cyclic handler or an alarm handler is actually started at the time according to the time resolution provided by the timer interrupt interval (`TTimPeriod`). For this reason, the cyclic handler is actually started at the time of timer interrupt occurrence immediately after the time when the cyclic handler should be started. A general T-Kernel implementation checks if a cyclic handler or an alarm handler that is to be started within the processing of timer interrupt exists, and then starts them as necessary.

4.7.2.1 tk_cre_cyc - Create Cyclic Handler

C Language Interface

```
#include <tk/tkernel.h>
```

ID cycid = tk_cre_cyc (CONST T_CCYC *pk_ccyc);

Parameter

CONST T_CCYC*	pk_ccyc	Packet to Create Cyclic Handler	Cyclic handler definition information

pk_ccyc Detail:

void*	exinf	Extended Information	Extended information
ATR	cycatr	Cyclic Handler Attribute	Cyclic handler attribute
FP	cychdr	Cyclic Handler Address	Cyclic handler address
RELTIM	cyctim	Cycle Time	Interval of cyclic start (ms)
RELTIM	cycphs	Cycle Phase	Cycle phase (ms)
UB	dsname[8]	DS Object name	DS object name

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

Return Parameter

ID	cycid	Cyclic Handler ID	Cyclic handler ID
	or	Error Code	Error code

Error Code

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, or cycphs is invalid or cannot be used)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Creates a cyclic handler, assigning to it a cyclic handler ID. This is performed by assigning a control block for the generated cyclic handler.

A cyclic handler is a handler running at specified intervals as a task-independent portion.

exinf can be used freely by the user to set 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 may need 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 kernel pays no attention to the contents of exinf.

cycatr indicates system attributes in its lower bits and implementation-dependent attributes in its higher bits. The system attribute part of cycatr is as follows.

```
cycatr := (TA_ASM || TA_HLNG) | [TA_STA] | [TA_PHS] | [TA_DSNAME]
```

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_DSNAME	Specifies DS object name

```
#define TA_ASM          0x00000000      /* assembly language 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_DSNAME       0x00000040      /* DS object name */
```

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 a function. The cyclic handler takes the following format when the TA_HLNG attribute is specified.

```
void cychdr( void *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 in a starting parameter.

cycphs indicates the length of time until the cyclic handler 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 in cycphs, the cyclic handler starts immediately after it is created. Zero cannot be specified in cyctim.

The starting of the cyclic handler for the nth time occurs after at least cycphs + cyctim * (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.

Even if a system call is invoked from a cyclic handler and this causes the task in RUNNING state up to that time to go to another state, with a different task going to RUNNING state, dispatching (task switching) does not occur while the cyclic handler is running. Completion of execution by the cyclic handler has precedence even if dispatching is necessary; only when the cyclic handler terminates does the dispatch take place. In other words, a dispatch request that is generated while a cyclic handler is running is not processed immediately, but is delayed until the cyclic handler terminates. This is called delayed dispatching.

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 WAITING state, or one that is intended for the invoking task.

When TA_DSNAME is specified, dsname is valid and specifies the DS object name. DS object name is used to identify objects by debugger, and it is handled only by T-Kernel/DS API, td_ref_dsname and td_set_dsname. For more details, see the description of td_ref_dsname and td_set_dsname. If TA_DSNAME is not specified, dsname is ignored. Then td_ref_dsname and td_set_dsname return E_OBJ error.

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 implementation-dependent whether to have them run serially (after one handler exits, another starts) or in a nested manner (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 portion, the principle of delayed dispatching applies.

If 0 is specified in cycphs, the first startup of the cyclic handler is executed immediately after this system call execution. However, depending on the implementation, the first startup (execution) of the cyclic handler may be executed while processing this system call, instead of immediately after the completion of this system call execution. In such case, the interrupt disabled or other state in the cyclic handler may differ from the state at the second and subsequent ordinary startups. In addition, when 0 is set to cycphs, the first startup of the cyclic handler is executed without waiting for a timer interrupt, that is, regardless of the timer interrupt interval. This behavior also differs from the second and subsequent startups of the cyclic handler, and from the startup of the cyclic handler with cycphs set to other than 0.

4.7.2.2 tk_cre_cyc_u - Create Cyclic Handler (in microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

ID cycid = tk_cre_cyc_u (CONST T_CCYC_U *pk_ccyc_u);

Parameter

CONST T_CCYC_U*	pk_ccyc_u	Packet to Create Cyclic Handler	Cyclic handler definition information

pk_ccyc_u Detail:

void*	exinf	Extended Information	Extended information
ATR	cycatr	Cyclic Handler Attribute	Cyclic handler attribute
FP	cychdr	Cyclic Handler Address	Cyclic handler address
RELTIM_U	cyctim_u	Cycle Time	Interval of cyclic start (microseconds)
RELTIM_U	cycphs_u	Cycle Phase	Cycle phase (microseconds)
UB	dsname[8]	DS Object name	DS object name

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

Return Parameter

ID	cycid		Cyclic Handler ID	Cyclic handler ID
		or	Error Code	Error code

Error Code

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_u, cychdr, cyctim_u, or cycphs_u is invalid or cannot be used)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

This system call takes 64-bit cyctim_u and cycphs_u in microseconds instead of the parameters cyctim and cycphs of tk_cre_cyc.

The specification of this system call is same as that of tk_cre_cyc, except that the parameter is replaced with cyctim_u and cycphs_u. For more details, see the description of tk_cre_cyc.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

4.7.2.3 tk_del_cyc - Delete Cyclic Handler

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_del_cyc (ID cycid);

Parameter

ID	cycid	Cyclic Handler ID	Cyclic handler ID

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Deletes a cyclic handler.

4.7.2.4 tk_sta_cyc - Start Cyclic Handler

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_sta_cyc (ID cycid);

Parameter

ID	cycid	Cyclic Handler ID	Cyclic handler ID

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

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.

4.7.2.5 tk_stp_cyc - Stop Cyclic Handler

C Language Interface

```
#include <tk/tkernel.h>

ER ercd = tk_stp_cyc (ID cycid );
```

Parameter

ID	cycid	Cyclic Handler ID	Cyclic handler ID

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Deactivates a cyclic handler, putting it in inactive state. It the cyclic handler was already in inactive state, this system call has no effect (no operation).

4.7.2.6 tk_ref_cyc - Reference Cyclic Handler Status

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_cyc (ID cycid , T_RCYC *pk_rcyc);

Parameter

ID	cycid	Cyclic Handler ID	Cyclic handler ID
T_RCYC*	pk_rcyc	Packet to Return Cyclic Handler Status	Pointer to the area to return the cyclic handler status

Return Parameter

ER	ercd	Error Code	Error code

pk_rcyc Detail:

void*	exinf	Extended Information	Extended information
RELTIM	lfttim	Left Time	Time remaining until the next handler starts (ms)
UINT	cycstat	Cyclic Handler Status	Cyclic handler activation state

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

Error Code

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 (invalid pk_rcyc)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

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 */
```

lfttim returns the remaining time (milliseconds) until the next time when the cyclic handler is invoked. It does not matter whether the cyclic handler is currently running or stopped.

`exinf` returns the extended information specified as a parameter when the cyclic handler is generated.`exinf` is passed to the cyclic handler as a parameter.

If the cyclic handler specified in `cycid` does not exist for, error code E_NOEXS is returned.

The time remaining `lfttim` returned in the cyclic handler status information (T_RCYC) is a value rounded to milliseconds. To know the value in microseconds, call tk_ref_cyc_u.

4.7.2.7 tk_ref_cyc_u - Reference Cyclic Handler Status (Microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_cyc_u (ID cycid , T_RCYC_U *pk_rcyc_u);

Parameter

ID	cycid	Cyclic Handler ID	Cyclic handler ID
T_RCYC_U*	pk_rcyc_u	Packet to Return Cyclic Handler Status	Pointer to the area to return the cyclic handler status

Return Parameter

ER	ercd	Error Code	Error code

pk_rcyc_u Detail:

void*	exinf	Extended Information	Extended information
RELTIM_U	lfttim_u	Left Time	Time remaining until the next handler starts (microseconds)
UINT	cycstat	Cyclic Handler Status	Cyclic handler activation state

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

Error Code

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 (invalid pk_rcyc_u)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

This system call takes 64-bit lfttim_u in microseconds instead of the return parameter lfttim of tk_ref_cyc.

The specification of this system call is same as that of tk_ref_cyc, except that the return parameter is replaced with lfttim_u. For more details, see the description of tk_ref_cyc.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

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. The ID number for an alarm handler is 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 system call is executed. When tk_stp_alm is called deactivating the alarm handler, the alarm time setting is canceled. Likewise, when an alarm time arrives and the alarm handler runs, the alarm time is canceled and the alarm handler becomes inactive.

Additional Notes
An alarm handler is actually started at the time according to the time resolution provided by the timer interrupt interval (`TTimPeriod`). For more details, see the additional notes for Section 4.7.2, “Cyclic Handler”.

4.7.3.1 tk_cre_alm - Create Alarm Handler

C Language Interface

```
#include <tk/tkernel.h>
```

ID almid = tk_cre_alm (CONST T_CALM *pk_calm);

Parameter

CONST T_CALM*	pk_calm	Packet to Create Alarm Handler	Alarm handler definition information

pk_calm Detail:

void*	exinf	Extended Information	Extended information
ATR	almatr	Alarm Handler Attribute	Alarm handler attributes
FP	almhdr	Alarm Handler Address	Alarm handler address
UB	dsname[8]	DS Object name	DS object name

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

Return Parameter

ID	almid		Alarm Handler ID	Alarm handler ID
		or	Error Code	Error code

Error Code

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 (pk_calm or almhdr is invalid or cannot be used)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Creates an alarm handler, assigning to it an alarm handler ID. This is performed by assigning a control block for the generated alarm handler.

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 set 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 may need 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 kernel pays no attention to the contents of exinf.

almatr indicates system attributes in its lower bits and implementation-dependent attributes in its higher bits. The system attribute part of almatr is as follows.

```
almatr := (TA_ASM || TA_HLNG) | [TA_DSNAME]
```

TA_ASM	The handler is written in assembly language
TA_HLNG	The handler is written in high-level language
TA_DSNAME	Specifies DS object name

```
#define TA_ASM          0x00000000      /* assembly language program */
#define TA_HLNG         0x00000001      /* high-level language program */
#define TA_DSNAME       0x00000040      /* DS object name */
```

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 a function. The alarm handler takes the following format when the TA_HLNG attribute is specified.

```
void almhdr( void *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 in a starting parameter.

Even if a system call is invoked from an alarm handler and this causes the task in RUNNING state up to that time to go to another state, with a different task going to RUNNING state, dispatching (task switching) does not occur while the alarm handler is running. Completion of execution by the alarm handler has precedence even if dispatching is necessary; only when the alarm handler terminates does the dispatch take place. In other words, a dispatch request that is generated while an alarm handler is running is not processed immediately, but is delayed until the alarm handler terminates. This is called delayed dispatching.

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 WAITING state, or one that is intended for the invoking task.

When TA_DSNAME is specified, dsname is valid and specifies the DS object name. DS object name is used to identify objects by debugger, and it is handled only by T-Kernel/DS API, td_ref_dsname and td_set_dsname. For more details, see the description of td_ref_dsname and td_set_dsname. If TA_DSNAME is not specified, dsname is ignored. Then td_ref_dsname and td_set_dsname return E_OBJ error.

Additional Notes

When multiple time event handlers or interrupt handlers operate at the same time, it is an implementation-dependent whether to have them run serially (after one handler exits, another starts) or in a nested manner (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 portion, the principle of delayed dispatching applies.

4.7.3.2 tk_del_alm - Delete Alarm Handler

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_del_alm (ID almid);

Parameter

ID	almid	Alarm Handler ID	Alarm handler ID

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Deletes an alarm handler.

4.7.3.3 tk_sta_alm - Start Alarm Handler

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_sta_alm (ID almid , RELTIM almtim);

Parameter

ID	almid	Alarm Handler ID	Alarm handler ID
RELTIM	almtim	Alarm Time	Alarm handler start relative time (ms)

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

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 anew with the alarm time specified here.

If almtim = 0 is set, the alarm handler starts as soon as it is activated.

4.7.3.4 tk_sta_alm_u - Start Alarm Handler (in microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_sta_alm_u (ID almid , RELTIM_U almtim_u);

Parameter

ID	almid	Alarm Handler ID	Alarm handler ID
RELTIM_U	almtim_u	Alarm Time	Alarm handler start relative time (microseconds)

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

This system call takes 64-bit almtim_u in microseconds instead of the parameter almtim of tk_sta_alm.

The specification of this system call is same as that of tk_sta_alm, except that the parameter is replaced with almtim_u. For more details, see the description of tk_sta_alm.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

4.7.3.5 tk_stp_alm - Stop Alarm Handler

C Language Interface

```
#include <tk/tkernel.h>

ER ercd = tk_stp_alm (ID almid );
```

Parameter

ID	almid	Alarm Handler ID	Alarm handler ID

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Cancels the alarm time of the alarm handler specified in almid, putting it in inactive state. It the cyclic handler was already in inactive state, this system call has no effect (no operation).

4.7.3.6 tk_ref_alm - Reference Alarm Handler Status

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_alm (ID almid , T_RALM *pk_ralm);

Parameter

ID	almid	Alarm Handler ID	Alarm handler ID
T_RALM*	pk_ralm	Packet to Return Alarm Handler Status	Pointer to the area to return the alarm handler status

Return Parameter

ER	ercd	Error Code	Error code

pk_ralm Detail:

void*	exinf	Extended Information	Extended information
RELTIM	lfttim	Left Time	Time remaining until the handler starts (ms)
UINT	almstat	Alarm Handler Status	Alarm handler activation state

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

Error Code

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 (invalid pk_ralm)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

References the status of the alarm handler specified in almid, 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    0x00 /* alarm handler is inactive */
#define TALM_STA        0x01    0x01 /* alarm handler is active */
```

If the alarm handler is active (TALM_STA), the relative time until the alarm handler is scheduled to be started next time is returned to lfttim. 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.

`exinf` returns the extended information specified as a parameter when the alarm handler is generated. `exinf` is passed to the alarm handler as a parameter.

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.

The time remaining `lfttim` returned in the alarm handler status information (T_RALM) is a value rounded to milliseconds. To know the value in microseconds, call tk_ref_alm_u.

4.7.3.7 tk_ref_alm_u - Reference Alarm Handler Status (Microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_alm_u (ID almid , T_RALM_U *pk_ralm_u);

Parameter

ID	almid	Alarm Handler ID	Alarm handler ID
T_RALM_U*	pk_ralm_u	Packet to Return Alarm Handler Status	Pointer to the area to return the alarm handler status

Return Parameter

ER	ercd	Error Code	Error code

pk_ralm_u Detail:

void*	exinf	Extended Information	Extended information
RELTIM_U	lfttim_u	Left Time	Time remaining until the handler starts (microseconds)
UINT	almstat	Alarm Handler Status	Alarm handler activation state

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

Error Code

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 (invalid pk_ralm_u)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

This system call takes 64-bit lfttim_u in microseconds instead of the return parameter lfttim of tk_ref_alm.

The specification of this system call is same as that of tk_ref_alm, except that the return parameter is replaced with lfttim_u. For more details, see the description of tk_ref_alm.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

4.8 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 WAITING state cannot be issued. Error code E_CTX is returned in such cases.

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.

4.8.1 tk_def_int - Define Interrupt Handler

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_def_int (UINT dintno , CONST T_DINT *pk_dint);

Parameter

UINT	dintno	Interrupt Handler Number	Interrupt handler number
CONST T_DINT*	pk_dint	Packet to Define Interrupt Handler	Interrupt handler definition information

pk_dint Detail:

ATR	intatr	Interrupt Handler Attribute	Interrupt handler attribute
FP	inthdr	Interrupt Handler Address	Interrupt handler address

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

Return Parameter

ER	ercd	Error Code	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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

"Interrupts" include both external interrupts from devices and interrupts due to CPU exceptions.

Defines an interrupt handler for the interrupt handler number dintno to enable use of the interrupt handler. This system call maps the interrupt handler number indicated in dintno to the address and attributes of the interrupt handler.

dintno is the number used to distinguish between different interrupt handlers. Its specific meaning is defined for each implementation, but generally the interrupt vector defined by the interrupt handling in the CPU hardware is used as it is, or any number that can be mapped to the interrupt vector is used. To get the interrupt handler number dintno from the interrupt vector, use the T-Kernel/SM DINTNO().

intatr indicates system attributes in its lower bits and implementation-dependent attributes in its higher bits. The system attribute part of intatr is specified as follows.

```
intatr := (TA_ASM || TA_HLNG)
```

TA_ASM	The handler is written in assembly language
TA_HLNG	The handler is written in high-level language

```
#define TA_ASM          0x00000000      /* assembly language program */
#define TA_HLNG         0x00000001      /* high-level language program */
```

As a rule, the kernel is not involved in the starting of a TA_ASM attribute interrupt handler. When an interrupt is raised, the interrupt handling function in the CPU hardware directly starts the interrupt handler defined by this system call (depending on the implementation, processing by program may be included). 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 an equivalent mechanism).

Support of a mechanism for return from an interrupt handler without using tk_ret_int and hence without kernel intervention is mandatory. Note that if tk_ret_int is not used, delayed dispatching does not need to be performed.

Support for return from an interrupt handler using tk_ret_int is also mandatory, and in this case delayed dispatching must be performed.

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 return from a C language function. The interrupt handler takes the following format when the TA_HLNG attribute is specified.

```
void inthdr( UINT dintno )
{
        /*
                Interrupt Handling
        */

        return; /* Exit interrupt handler */
}
```

The parameter dintno passed to an interrupt handler is the interrupt handler 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, multiple interrupts are disabled, and this state remains when the interrupt handler is called. If multiple interrupts are to be allowed, the interrupt handler must include processing that handles multiple interrupts by manipulating the CPU interrupt flag.

Also in the case of the TA_HLNG attribute, upon entry into the interrupt handler, issuing system call must be possible. Note, however, that assuming standard provision of the functionality described above, extensions are allowed such as adding a function for entering an interrupt handler with multiple interrupts enabled.

When the TA_ASM attribute is specified, the state upon entry into the interrupt handler shall be defined for each implementation. Such matters as the stack and register status 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 kernel 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 detected whether it runs as task-independent portion, that method shall be announced for each implementation.

Whether the TA_HLNG or TA_ASM attribute is specified, upon entry into an interrupt handler, the logical memory space at the time the interrupt occurred is retained. No processing takes place upon return from the interrupt handler for restoring the logical memory space to its state at the time the interrupt was raised. Switching logical memory spaces inside the interrupt handler is not prohibited, but the kernel is not aware of the effect of such logical memory space switching.

Even if a system call is invoked from an interrupt handler and this causes the task in RUNNING state up to that time to go to another state, with a different task going to RUNNING state, dispatching (task switching) does not occur while the interrupt handler is running. Completion of execution of the interrupt handler has precedence even if dispatching is necessary; only when the interrupt handler terminates does the dispatch take place. In other words, a dispatch request that is generated while an interrupt handler is running is not processed immediately, but is delayed until the interrupt handler terminates. This is called delayed dispatching.

An interrupt handler runs as a task-independent portion. As such, it is not possible to call in an interrupt handler a system call that can enter WAITING 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 definitions are canceled, the default handler defined by the system is used.

It is possible to redefine an interrupt handler for an interrupt handler number that is already defined. 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 error.

Additional Notes

The various specifications governing the TA_ASM attribute are mainly concerned with realizing an interrupt hook. For example, when an exception is raised due to 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, the default interrupt handler defined by the system may perform the processing and starts a debugger. In this case, the interrupt handler defined by high-level program hooks the default interrupt handler defined by the system. And, according to the situation, the handler either passes the interrupt handling to a system program such as a debugger, or it just processes it for itself.

4.8.2 tk_ret_int - Return from Interrupt Handler

C Language Interface

```
#include <tk/tkernel.h>
```

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.

Parameter

None

Return Parameter

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. The behavior in case an error occurs is implementation-dependent.

E_CTX	Context error (issued from other than an interrupt handler (implementation-dependent error))

Valid Context

Task portion	Quasi-task portion	Task-independent portion
NO	NO	YES

Description

Exits from an interrupt handler.

System calls invoked from an interrupt handler do not result in dispatching while the handler is running; instead, the dispatching is delayed until tk_ret_int is called ending the interrupt handler processing(delayed dispatching). Accordingly, tk_ret_int results in the processing of all dispatch requests made while the interrupt handler was running.

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 kernel 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 CPU hardware interrupt processing function. The saving and restoring of registers used by the interrupt handler must therefore be taken care of 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 which case tk_ret_int can be implemented using a trap instruction of another vector separate 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 these system calls do not return even if error is detected.

Using an assembly language return-from-interrupt 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 implementing the kernel, it may be possible to perform delayed dispatching even when an interrupt handler exits using an assembly language return-from-interrupt instruction. In such cases, it is permissible for the assembly language return-from-interrupt instruction to be interpreted as if 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 implementation, control may return from a different type of handler immediately.

4.9 System Management Functions

System management functions sets and references system states. Functions are provided for rotating task precedence in a queue, getting the ID of the task in RUNNING state, disabling and enabling task dispatching, referencing context and system states, setting low-power mode, and referencing the T-Kernel version.

4.9.1 tk_rot_rdq - Rotate Ready Queue

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_rot_rdq (PRI tskpri);

Parameter

PRI	tskpri	Task Priority	Task priority

Return Parameter

ER	ercd	Error Code	Error code

Error Code

E_OK	Normal completion
E_PAR	Parameter error (tskpri is invalid)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

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.

By setting tskpri = TPRI_RUN = 0, this system call rotates the precedence of tasks having the priority level of the task currently in RUNNING state. When tk_rot_rdq is called from an ordinary task, it rotates the precedence of tasks having the same priority as the invoking task. When calling from a cyclic handler or other task-independent portion, it is also possible to call tk_rot_rdq (tskpri = TPRI_RUN).

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, the precedence of the invoking task will be the lowest among tasks of the same priority. This system call can therefore 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 4.10, “Precedence Before Issuing tk_rot_rdq” and Figure 4.11, “Precedence After Issuing tk_rot_rdq (tskpri = 2)”. When this system call is issued in the state

shown in Figure 4.10, “Precedence Before Issuing tk_rot_rdq” specifying `tskpri` = 2, the new precedence order becomes that in Figure 4.11, “Precedence After Issuing tk_rot_rdq (`tskpri` = 2)”, and Task C becomes the executing task.

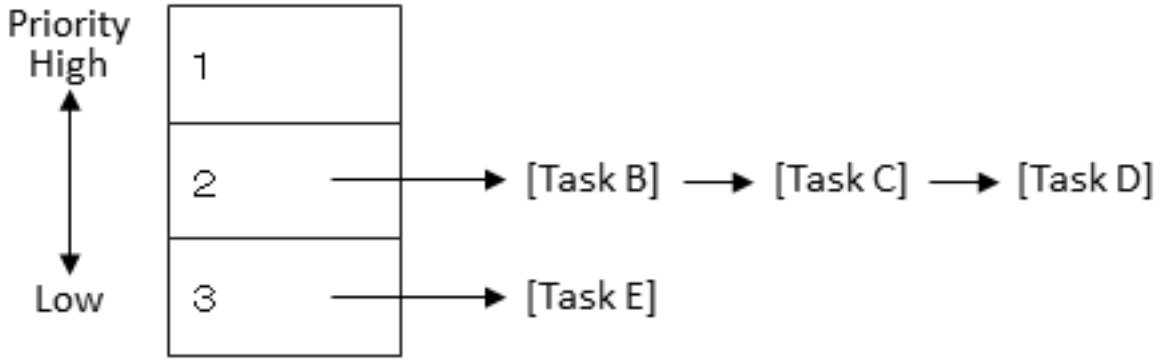

Figure 4.10: Precedence Before Issuing tk_rot_rdq

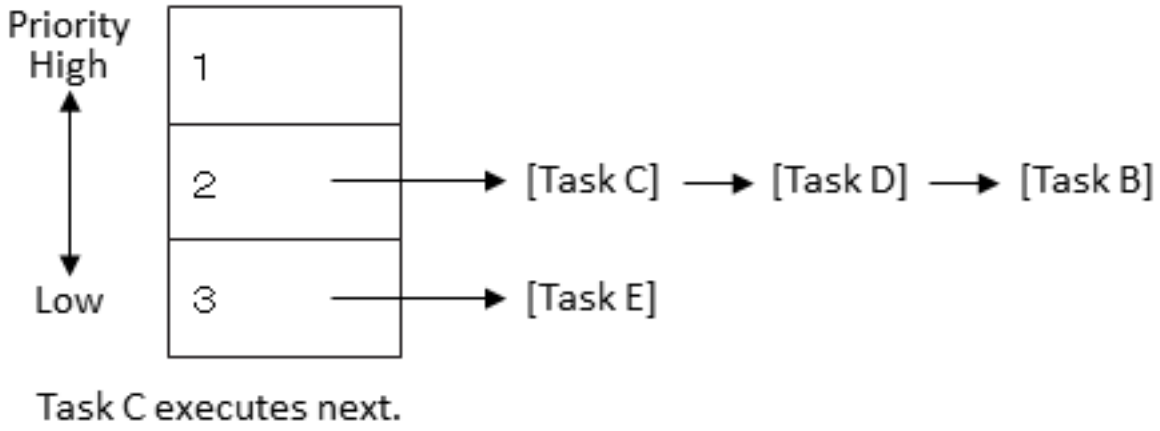

Figure 4.11: Precedence After Issuing tk_rot_rdq (`tskpri` = 2)

4.9.2 tk_get_tid - Get Task Identifier

C Language Interface

```
#include <tk/tkernel.h>

ID tskid = tk_get_tid ( void );
```

Parameter

None

Return Parameter

ID	`tskid`	Task ID	ID of the task in RUNNING state

Error Code

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the ID number of the task currently in RUNNING state. Unless the task-independent portion is executing, the current RUNNING state task will be the invoking task.

If there is no task currently in RUNNING state, 0 is returned.

Additional Notes

The task ID returned by tk_get_tid is identical to `runtskid` returned by tk_ref_sys.

4.9.3 tk_dis_dsp - Disable Dispatch

C Language Interface

```
#include <tk/tkernel.h>

ER ercd = tk_dis_dsp ( void );
```

Parameter

None

Return Parameter

ER	ercd	Error Code	Error code

Error Code

E_OK	Normal completion
E_CTX	Context error (issued from task-independent portion)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Disables task dispatching. Dispatch disabled state remains in effect until tk_ena_dsp is called enabling task dispatching. While dispatching is disabled, the invoking task does not change from RUNNING state to READY state or to WAITING 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. Dispatching of the higher-priority task is delayed until dispatch disabled state ends.
- If the task that called tk_dis_dsp issues a system call that may cause the invoking task to be put in WAITING state (e.g., tk_slp_tsk or tk_wai_sem), error code E_CTX is returned.
- 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 one time is enough to enable dispatching again. The sophisticated operation when the pair of system calls tk_dis_dsp and tk_ena_dsp are used in a nested manner must therefore be managed by the user as necessary.

Additional Notes

A task in RUNNING 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 RUNNING 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.

4.9.4 tk_ena_dsp - Enable Dispatch

C Language Interface

```
#include <tk/tkernel.h>

ER ercd = tk_ena_dsp ( void );
```

Parameter

None

Return Parameter

ER	ercd	Error Code	Error code

Error Code

E_OK	Normal completion
E_CTX	Context error (issued from task-independent portion)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 from a task not in dispatch disabled state, the dispatch enabled state continues and no error code is returned.

4.9.5 tk_ref_sys - Reference System Status

C Language Interface

```
#include <tk/tkernel.h>

ER ercd = tk_ref_sys (T_RSYS *pk_rsys );
```

Parameter

T_RSYS*	pk_rsys	Packet to Refer System Status	Pointer to the area to return the system status

Return Parameter

ER	ercd	Error Code	Error code

pk_rsys Detail:

INT	sysstat	System State	System State
ID	runtskid	Running Task ID	ID of the task currently in RUNNING state
ID	schedtskid	Scheduled Task ID	ID of the task scheduled to run next

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

Error Code

E_OK	Normal completion
E_PAR	Parameter error (invalid pk_rsys)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the current system execution status, passing in return parameters such information as the dispatch disabled state and whether a task-independent portion is executing.

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 is running
TSS_DDSP	1	Dispatch disabled
TSS_DINT	2	Interrupts disabled
TSS_INDP	4	Task-independent portion is running
TSS_QTSK	8	Quasi-task portion is running

The ID of the task currently in RUNNING state is returned in `runtskid`, while `schedtskid` indicates the ID of the next task scheduled to go to RUNNING 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 kernel implementation, the information returned by tk_ref_sys is not necessarily guaranteed to be accurate at all times.

4.9.6 tk_set_pow - Set Power Mode

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_set_pow (UINT powmode);

Parameter

UINT	powmode	Power Mode	Low-power mode

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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.

```
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 incremented. 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. For more details, see Section 5.6, “Power Management Functions”.

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.

4.9.7 tk_ref_ver - Reference Version Information

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_ver (T_RVER *pk_rver);

Parameter

T_RVER*	pk_rver	Packet to Return Version Information	Pointer to the area to return the version information

Return Parameter

ER	ercd	Error Code	Error code

pk_rver Detail:

UH	maker	Maker Code	T-Kernel maker code
UH	prid	Product ID	T-Kernel identification number
UH	spver	Specification Version	Specification version
UH	prver	Product Version	T-Kernel version
UH	prno[4]	Product Number	T-Kernel products management information

Error Code

E_OK	Normal completion
E_PAR	Parameter error (invalid pk_rver)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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 maker code of the T-Kernel implementing vendor. The maker field has the format shown in Figure 4.12, “maker Format”.

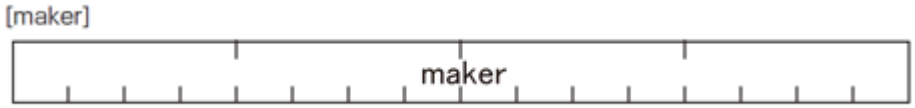

Figure 4.12: maker Format

prid is a number indicating the T-Kernel type. The prid field has the format shown in Figure 4.13, "prid Format".

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 original version of T-Kernel is provided from T-Engine Forum, and its maker and prid are as follows.

```
maker   = 0x0000
prid    = 0x0000
```

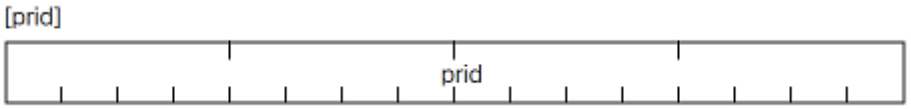

Figure 4.13: prid Format

The upper 4 bits of spver give the TRON specification series. The lower 12 bits indicate the T-Kernel specification version implemented. The spver field has the format shown in Figure 4.14, "spver Format".

If, for example, a product conforms to the T-Kernel specification Ver 2.01.xx, spver is as follows.

```
MAGIC   = 0x7           (T-Kernel)
SpecVer = 0x201         (Ver 2.01)
spver   = 0x7201
```

If a product implements the T-Kernel specification draft version Ver 2.B0.xx, spver is as follows.

```
MAGIC   = 0x7           (T-Kernel)
SpecVer = 0x2B0         (Ver 2.B0)
spver   = 0x72B0
```

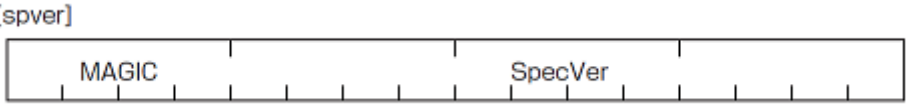

Figure 4.14: spver Format

MAGIC:
Type of OS specification

0x0	TRON common (TAD, etc.)
0x1	reserved
0x2	reserved
0x3	reserved
0x4	reserved
0x5	reserved
0x6	reserved
0x7	T-Kernel

`SpecVer`:
The version of the specification that 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.

`prver` is the version number of the T-Kernel implementation. The specific values assigned to `prver` are left to the T-Kernel implementing vendor to decide.

`prno` is a return parameter for use in indicating T-Kernel product management information, product number or the like. The specific meaning of values set in `prno` is left to the T-Kernel implementing vendor to decide.

Additional Notes

The format of the packet and structure members for getting version information is mostly uniform across the various T-Kernel specifications.

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.

A kernel 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 2.A1 → Ver 2.A2 → Ver 2.B1 → Ver 2.C1 → Ver 2.00 → Ver 2.01... In this example, when going from Ver 2.Cx to Ver 2.00, `SpecVer` goes from a higher to a lower value.

4.10 Subsystem Management Functions

Subsystem management functions extends the functions of T-Kernel itself by adding a user-defined function called "subsystem" to the kernel in order to implement middleware and others running on the T-Kernel. Some functions provided by T-Kernel/SM are also implemented by utilizing the subsystem management functions.

A subsystem consists of extended SVC handlers to execute user-defined system calls (called "extended SVCs"), a break function that performs the required processing when any exception occurs, an event handling function that performs the required processing when any event is raised from devices, etc., startup and cleanup functions that perform required processing at the start/exit of task for each resource group, and resource control blocks (Figure 4.15, “T-Kernel Subsystems”.)

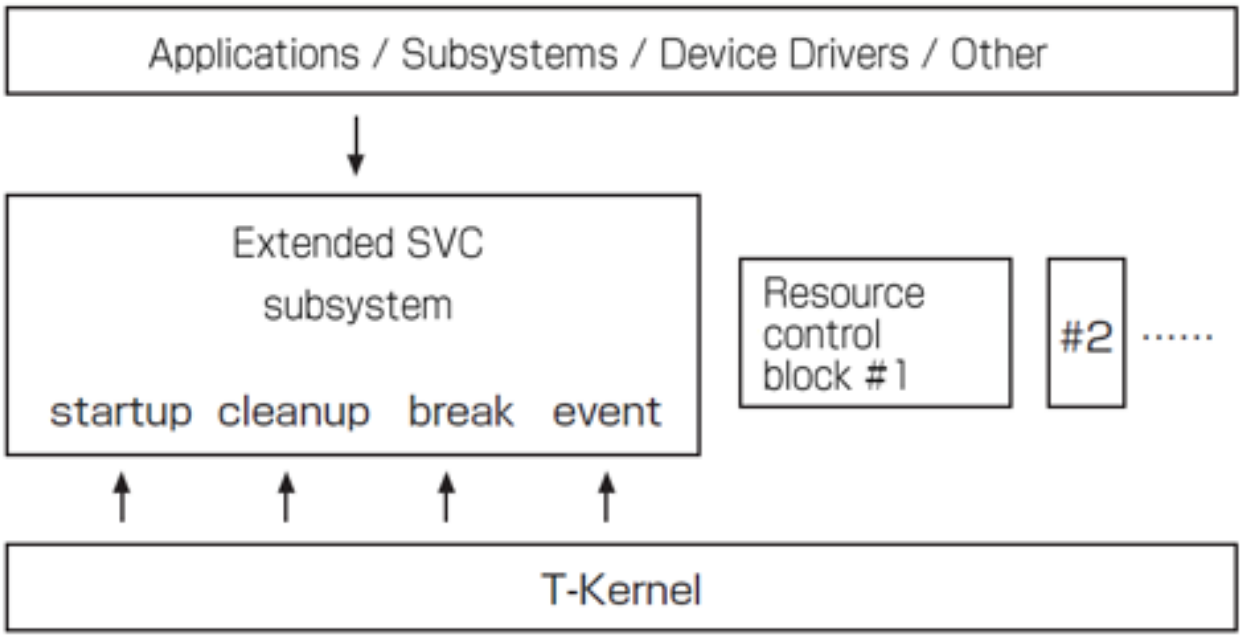

Figure 4.15: T-Kernel Subsystems

The extended SVC handler directly accepts requests from applications and others. A break function, event processing function, startup function, and cleanup function are so-called callback type functions and accept requests from the kernel.

Additional Notes

Functions of T-Kernel Extension (T-Kernel Standard Extension) including the process management functions and the file management functions are also implemented by utilizing the subsystem management functions. Other examples of middleware for T-Kernel that are implemented by utilizing the subsystem management functions include TCP/IP manager, USB manager, and PC card manager.

Though subsystem management functions are equivalent to the extended SVC handlers and extended service calls provided in ITRON specification, they can be used to build complex and advanced middleware through not only the addition of just user-defined system calls but also through provision of resource management functions and exception processing functions to handle the exceptions, which are required for the added system calls.

Subsystem management functions manage resources by each resource group to which the task belongs. T-Kernel Extension (T-Kernel Standard Extension), a high level middleware of T-Kernel, uses T-Kernel resource group functions to realize a process. Because of the relationship described above, the resource management can be performed independently for each process in a subsystem by automatic execution of starup function or cleanup function defined in the subsystem upon creation (starting) or termination of a process. For example, if you want to automatically close a file that is not closed at the time of process termination, you can do so in the cleanup function included in the file management subsystem.

In addition to the subsystem management functions, T-Kernel also provides the device driver functions in order to extend itself. Both subsystems and device drivers are function modules independent from T-Kernel itself. They can be used by loading their corresponding binary programs into system space and then calling them from a task on T-Kernel. Both run at the protection level 0. While API is limited to using open/close and read/write type when calling a device driver, API for calling a subsystem can be defined without any restriction. Moreover, for the subsystem, there is a function that automatically manages a resource at the time of creating (starting) or terminating a resource group (process), for the device driver, there is no function to do so.

Subsystems are identified by subsystem IDs (`ssid`), more than one subsystem can be defined and used at the same time. One subsystem can be called and used from within another subsystem.

4.10.1 tk_def_ssy - Define Subsystem

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_def_ssy (ID ssid , CONST T_DSSY *pk_dssy);

Parameter

ID	ssid	Subsystem ID	Subsystem ID
CONST T_DSSY*	pk_dssy	Packet to Define Subsystem	Subsystem definition information

pk_dssy Detail:

ATR	ssyatr	Subsystem Attributes	Subsystem attributes
PRI	ssypri	Subsystem Priority	Subsystem priority
FP	svchdr	Extended SVC Handler Address	Extended SVC handler address
FP	breakfn	Break Function Address	Break function address
FP	startupfn	Startup Function Address	Startup function address
FP	cleanupfn	Cleanup Function Address	Cleanup function address
FP	eventfn	Event Handling Function Address	Event handling function address
INT	resblksz	Resource Control Block Size	Resource control block size (in bytes)

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

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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 (ssyatr 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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Defines subsystem specified in `ssid`.

One subsystem ID must be assigned to one subsystem without overlapping with other subsystems. The kernel does not have a function for assigning subsystem IDs 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 lower bits and implementation-dependent attributes in its higher bits. The system attribute 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 calling order is undefined when these subsystems have the same priority. 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 resource control block for the subsystem of `ssid` will also be deleted.

- Resource control block

  The resource control block defines groups of resources and manages them by their attributes and other factors. Resource control block is allocated for each resource group. The block has its own memory area of the size specified in `resblksz`. If `resblksz` = 0 is specified, no resource control block is allocated; but a resource ID (see tk_cre_res) is assigned even in this case.

  Each task belongs to one resource group. When a task makes a request to a subsystem and resources are allocated to that task in the subsystem, the allocation information is stored in the resource control block. The subsystem decides what kinds of resource information to register in the resource control block and how they are to be registered.

  The kernel is not responsible for the content of the resource control block; it can be used freely by the subsystem. The size specified in `resblksz` should, however, be as small as possible. If a larger memory block is needed, the subsystem should allocate that memory on its own and register its address in the resource control block.

  A resource control block is located in resident memory of shared (system) space.

- Extended SVC handler

  An extended SVC handler accepts requests from applications and other programs as an application programming interface (API) for a subsystem. 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( void *pk_para, FN fncd )
{
        /*
                branching by fncd
        */

        return retcode; /* exit extended SVC handler */
}
```

  `fncd` is a function code. The lower 8 bits of the instruction code are the subsystem ID. The remaining higher 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 from an extended SVC handler is passed to the caller transparently as the function return code. 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 error code (negative value) set by T-Kernel 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 kernel implementation. As a subsystem API, however, it must be specified in a C language function format independent of the kernel implementation. The subsystem must provide an interface library for converting from the C language function format to the kernel-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 role of a break function is to abort the processing of the currently running extended SVC handler.

  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 in which 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 the nesting level of the extended SVC handler is decreased by the return from the latest extended SVC handler, the break function corresponding to the former extended SVC handler to which the control will be returned next, is called.

  A break function is called only once 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 requesting task is identified as follows: If a break function is called by tk_ras_tex, it runs as a quasi-task portion of the task that issued tk_ras_tex. On the other hand, when the nesting level of extended SVC handler is decreased, the break function runs as a quasi-task portion of the task that raised the task exception (the task running the extended SVC handler). This means that the task executing the break function may 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 waiting 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 RUNNING 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.

In the case where the requesting task of the break function differs from that of the extended SVC handler, that is, where the break function and the extended SVC handler run in different task contexts, the task priority of the break function is raised to the same as that of the extended SVC handler only while the break handler is executing if the former is lower than the latter. 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 immediately before entry into the break function; any changes after that of the extended SVC handler task priority are not followed 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 results in a priority change in the extended SVC handler task. At the same time, there is no restriction on priority changes due to a running break function.

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 specified in 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, the extended SVC handler returns error code when the API is called and cannot be executed normally, as a result of unsuccessful initialization of the resource control block.

  A startup function runs as a quasi-task portion of the task that issued tk_sta_ssy.

- Cleanup function

  A cleanup function is called by issuing the tk_cln_ssy system call.

  It 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 to be released, while `info` is a parameter that can be used freely. Both are parameters specified in tk_cln_ssy.

  Even if releasing fails for some reason, the cleanup function must be terminated normally. The error handling method, such as logging of errors, are left to the subsystem implementing vendor to decide.

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 of the task that issued 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 has 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. For more details, see Section 5.3, "Device Management Functions".

```
#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 of the task that issued tk_evt_ssy.

Additional Notes

Extended SVC handlers as well as break functions, startup functions, cleanup functions and event handling functions all have the 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 the startup function, and after resource release by the 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 kernel does not make any attempt to prevent this kind of call. Basically it is necessary to avoid calling an extended SVC before calling the startup function and after calling the cleanup function.

There may be cases where, for some reason or other, the break function, cleanup function or event handling function is called without first calling the 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.

The task space in the extended SVC handler is the same as that of the caller. Therefore, it is not necessary to switch the task space even when accessing the buffer passed by the caller. However, the extended SVC

handler runs at protection level 0 (privileged mode), which makes it possible to access the memory that the caller task is not permitted to access. For this reason, in the extended SVC handler, the access permission check should be performed as necessary, using ChkSpaceR(), ChkSpaceRW(), and so on.

It is possible to issue a system call that enters WAITING state in the extended SVC handler, but in that case the program must be designed so that it can be stopped by calling a break function. The specific processing flow is as follows: If tk_ras_tex is issued for the caller task while an extended SVC handler is executing, it is necessary to stop the running extended SVC handler as soon as possible and return a stop error to the caller task. For this purpose the break function is used. In order to stop the running extended SVC handler immediately, the break function must forcibly release the WAITING state, even if the system call is in WAITING state during processing the extended SVC handler. For this purpose, the tk_dis_wai system call is generally used. tk_dis_wai can prevent the system call from entering WAITING state until the control returns from the extended SVC handler to the caller task, but the implementor should also make it possible to stop the program of the extended SVC handler by calling a break function. For example, leaving from WAITING state with the error code E_DISWAI can mean that the execution is stopped by a break function. So it is best to stop the extended SVC handler immediately and return a stop error to the caller task, without continuing to execute the subsequent processing.

An extended SVC handler may be called concurrently by multiple tasks. If the tasks share same resources, the mutual exclusion control must be performed in the extended SVC handler.

4.10.2 tk_sta_ssy - Call Startup Function

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_sta_ssy (ID ssid , ID resid , INT info);

Parameter

ID	ssid	Subsystem ID	Subsystem ID
ID	resid	Resource ID	Resource ID
INT	info	Information	Any parameter

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Calls the startup function of the subsystem specified in ssid.

Specifying ssid = 0 makes the system call applied to all currently defined subsystems. In this case the startup function of each subsystem is called in descending order of priority.

The calling order is undefined when these subsystems have the same priority.

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.

If this system call is issued for a subsystem with no startup function defined, the function is simply not called; no error results.

If a task exception is raised for the task that called tk_sta_ssy during startup function execution execution, the task exception is held until the startup function completes its processing.

Additional Notes

T-Kernel Extension (T-Kernel Standard Extension), a higher level middleware of T-Kernel, uses tk_sta_ssy and tk_cln_ssy to perform the startup processing during process creation (startup) and the cleanup processing

during process termination, respectively. Specifically, during the processing of process creation (startup) in T-Kernel Extension, tk_sta_ssy is issued specifying `ssid` = 0 to perform the startup processing for the newly started process. During the processing of process termination in T-Kernel Extension, tk_cln_ssy is issued specifying `ssid` = 0 to perform the cleanup processing for the process to be terminated. For example, when the file management subsystem performs the cleanup processing for terminating a process, the subsystem can use this function to automatically close the file opened by that process.

If multiple subsystems are defined, the startup/cleanup function of each subsystem is executed in the order determined by subsystem priority, which is reversed between the startup processing and the cleanup processing.

For example, in the case where Subsystem A is used to implement another Subsystem B, the priority of Subsystem A should be higher than that of Subsystem B. This makes the startup processing of Subsystem A being executed before Subsystem B for the process to be newly started. Thus, the function (extended SVC handler) of Subsystem A can be called during the startup processing of Subsystem B. On the other hand, the cleanup processing of Subsystem B is executed before Subsystem A for the process to be terminated. Thus, the function (extended SVC handler) of Subsystem A can be called during the cleanup processing of Subsystem B (see Figure 4.16, "Dependency and Priority of Subsystems").

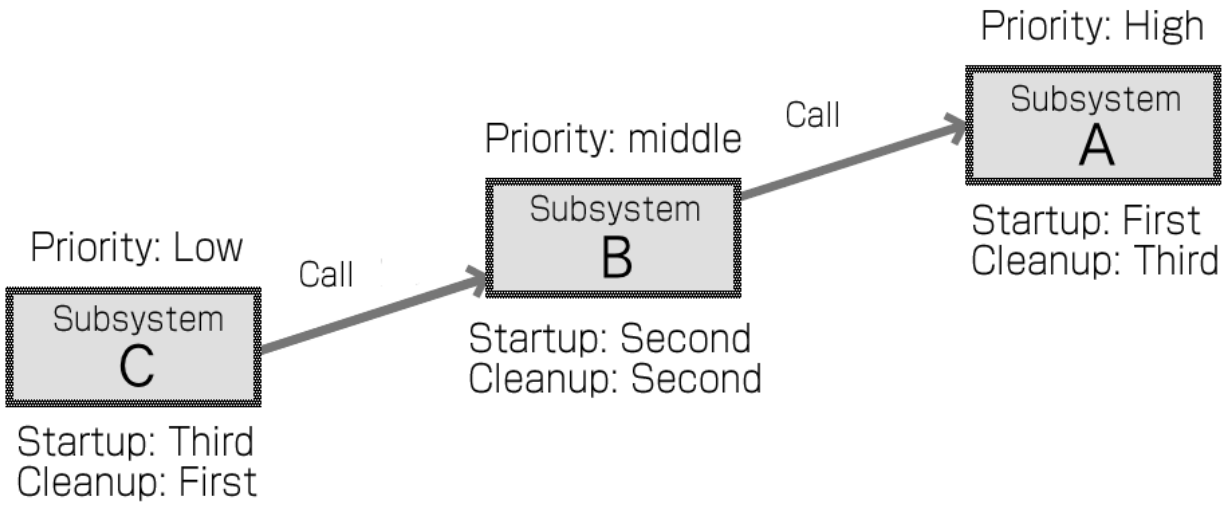

Figure 4.16: Dependency and Priority of Subsystems

The startup functions of all the subsystems are always executed each time a new process is created (started). The started process does not necessarily use all of the subsystem functions, or it may never call them. Considering that all of the startup functions of subsystems are executed when a process (including one unrelated to the subsystems) is created (started), the overhead due to startup functions should be minimized. To do this, the startup function should only perform the bare minimum of processing, and a complicated processing, if necessary, should be deferred without being executed in the startup function until the subsystem is actually used, for example when the extended SVC handler is called from the process for the first time.

4.10.3 tk_cln_ssy - Call Cleanup Function

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_cln_ssy (ID ssid , ID resid , INT info);

Parameter

ID	ssid	Subsystem ID	Subsystem ID
ID	resid	Resource ID	Resource ID
INT	info	Information	Any parameter

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Calls the cleanup function of the subsystem specified in ssid.

Specifying ssid = 0 makes the system call applied to all currently defined subsystems. In this case the cleanup function of each subsystem is called in ascending order of priority.

The calling order is undefined when these subsystems have the same priority.

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.

If this system call is issued for a subsystem with no cleanup function defined, the function is simply not called; no error results.

If a task exception is raised for the task that called tk_cln_ssy during cleanup function execution, the task exception is held until the cleanup function completes its processing.

4.10.4 tk_evt_ssy - Call Event Function

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_evt_ssy (ID ssid , INT evttyp , ID resid , INT info);

Parameter

ID	ssid	Subsystem ID	Subsystem ID
INT	evttyp	Event Type	Event request type
ID	resid	Resource ID	Resource ID
INT	info	Information	Any parameter

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)
Other	Error code returned by the event handling function

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Calls the event handling function of the subsystem specified in ssid.

Specifying ssid = 0 makes the system call applied 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 subsystems in descending order of priority.

When evttyp is an even number:
Calls subsystems in ascending order of priority.

The calling order is undefined wheren these subsystems have the same priority.

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, set resid to 0.

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 a task exception is raised for the task that called tk_evt_ssy, during the execution of event handling function, the task exception is held until the event handling function completes its processing.

Additional Notes

An example of using an event handling function is to perform the suspend/resume processing for the power management functions. Specifically, when the system enters the power-off state (device suspended state) due to power failure or other reason, it notifies each subsystem of its transition to suspended state. Then the event handling function of each subsystem is called to perform the appropriate processing for it. In T-Kernel/SM, tk_evt_ssy is executed for this purpose during the processing of tk_sus_dev. The event handling function of each subsystem performs any necessary operations before going to suspended state, such as saving the data. On the other hand, when the system returns (resumes) from the suspended state due to power on or other reason, it notifies each subsystem of its return from suspended state. Then the event handling function of each subsystem is called again to perform the appropriate processing for it. For more details, see the description of tk_sus_dev.

For another example, when a new device is registered by tk_def_dev, the system notifies each subsystem of the registration, and the event handling function of each subsystem is called to perform the appropriate processing for it. In T-Kernel/SM, tk_evt_ssy is executed for this purpose during the processing of tk_def_dev.

4.10.5 tk_ref_ssy - Reference Subsystem Status

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_ssy (ID ssid , T_RSSY *pk_rssy);

Parameter

ID	ssid	Subsystem ID	Subsystem ID
T_RSSY*	pk_rssy	Packet to Return Subsystem Status	Pointer to the area to return the subsystem definition information

Return Parameter

ER	ercd	Error Code	Error code

pk_rssy Detail:

PRI	ssypri	Subsystem Priority	Subsystem priority
INT	resblksz	Resource Control Block Size	Resource control block size (in bytes)

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

Error Code

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 (invalid pk_rssy)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

References information about the subsystem specified in ssid.

ssypri returns the subsystem priority specified in tk_def_ssy.

resblksz returns the size of the resource control block specified in tk_def_ssy.

If the subsystem specified in ssid is not defined, E_NOEXS is returned.

4.10.6 tk_cre_res - Create Resource Group

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_cre_res (void);

Parameter

None

Return Parameter

ID	resid		Resource ID	Resource ID
		or	Error Code	Error code

Error Code

E_LIMIT	Number of resource groups exceeds the system limit
E_NOMEM	Insufficient memory (memory for control block cannot be allocated)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Creates a new resource group, assigning to it 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 (see the description of Figure 4.17, “Subsystems and Resource Groups”).

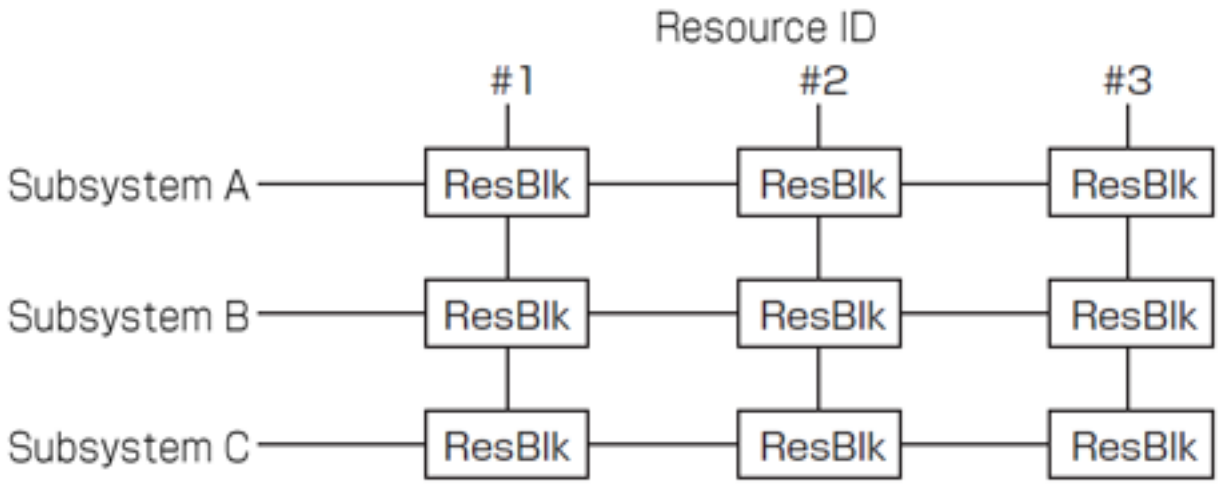

Figure 4.17: Subsystems and Resource Groups

A new subsystem can 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 must be created by tk_def_ssy.

For example, if a new subsystem ID is defined in a situation like that shown in Figure 4.17, "Subsystems and Resource Groups", 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.

A system resource group always exists as a special resource group. One system resource group always exists, moreover, from the time the system boots, without waiting for creation by tk_cre_res. The system resource group cannot be deleted. Other than the point that it always exists, a system resource group is no different from other resource groups.

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 kernel implementation.

T-Kernel Extension (T-Kernel Standard Extension), a higher level middleware of T-Kernel, uses the resource group function of T-Kernel to achieve various functions of process, where one process corresponds to one resource group. For this reason, when creating (starting) a process, it is necessary to allocate a resource control block for it by executing tk_cre_res.

Using the resource control function, each subsystem can allocate an independent resource to each process (that is, to each resource group), or can automatically release the allocated resource when the process is

terminated. For example, a file management subsystem often assigns an identifier called "file descriptor" to a file each time a process opens it, and usually uses that file descriptor for subsequent file manipulations. In this case, various management information identified by the file descriptor for file manipulation is the resource. Placing this resource in the resource control block for the file management subsystem allows the information for file manipulation to be managed independently for each process (resource group).

Generally, for subsystems that realize functions which should be controlled independently for each process, it is effective to use the resource control block to manage the information of each process independently. It is also possible to use the startup function to prepare the subsystem side or initialize the resource control block for a newly created (started) process, or to use the cleanup function to automatically release the resources when the process is terminated. On the other hand, for subsystems that realize functions which is not directly related to a process (such as functions shared between processes, or functions for the entire system), functions related to the resource control block, resources, and resource groups have less chance to be used.

When a new process is created (started), the resource control block for each subsystem is allocated in the resident memory area of the system shared space, regardless of whether the process actually uses the subsystem or not. That means that some system shared memory is consumed. To reduce the overhead for the entire system, it is best to minimize the size of the resource control block.

Suppose, for example, there is a subsystem that needs 1 MB of independent working memory for each process. As the working memory is required for each process, you might choose to use a part of the resource control block for the working memory, but the amount is too large for the resource control block. If the resource control block size is set to 1 MB, that amount of space is unconditionally allocated each time when a new process is created (started), which consumes too much resident memory of the system shared space. Especially, if a new process never uses the function of this subsystem, too much memory is wasted.

In such case, it is best to defer the allocation of the working memory used by subsystem until it is actually required. To do so, for example, include in the resource control block only the flag indicating whether the working memory space has been allocated or not, and the address of the working memory space. Then check the flag when the process uses the subsystem function (calls the extended SVC handler), and allocate the working memory space only if it is not yet allocated. This solution can eliminate the waste of memory space caused by allocating the large resource control block to a process that does not call the subsystem.

4.10.7 tk_del_res - Delete Resource Group

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_del_res (ID resid);

Parameter

ID	resid	Resource ID	Resource ID

Return Parameter

ER	ercd	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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).

4.10.8 tk_get_res - Get Resource Management Block

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_get_res (ID resid , ID ssid , void **p_resblk);

Parameter

ID	resid	Resource ID	Resource ID
ID	ssid	Subsystem ID	Subsystem ID
void**	p_resblk	Resource Control Block	Pointer to the area to return the return parameter resblk

Return Parameter

void*	resblk	Resource Control Block	Resource control block
ER	ercd	Error Code	Error code

Error Code

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 (invalid p_resblk)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

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.

Chapter 5

T-Kernel/SM Functions

This chapter describes details of the functions provided by T-Kernel/SM (System Manager).

Overall Note and Supplement

- There are two types of API names that are defined in T-Kernel/SM specification: one beginning with tk_ and others. As a general rule, APIs with a name beginning with tk_ are implemented in extended SVC, and other APIs are implemented as library functions (including in-line functions) or macros of the C language. APIs that are defined in T-Kernel/SM, however, are not called as "system call." The word "system call" refers to APIs that are defined in T-Kernel/OS or T-Kernel/DS.
- Some libraries and macros call some extended SVC or system calls indirectly.
- Error codes such as E_PAR, E_MACV, and E_NOMEM that can be returned in many situations are not described here always unless there is some special reason for doing so.
- Except where otherwise noted, extended SVC and libraries of T-Kernel/SM cannot 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 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).
- Extended SVC and libraries of T-Kernel/SM are reentrant except when a special explanation is given. Note that some functions perform mutual exclusion internally.

5.1 System Memory Management Functions

The system memory management functions are for managing all the memory (system memory) allocated dynamically by T-Kernel. This includes memory used internally by T-Kernel as well as task stacks, message buffers, and memory pools.

System memory is managed in memory block units. A block size is usually a page size defined in MMU, and assumed to be approximately 4 KB in the current implementation. A system that does not use an MMU can set any desired block size, but approximately same size as the MMU page size is recommended. Block size can be retrieved by calling tk_ref_smb.

System memory is allocated in the system space. T-Kernel does not manage task space memory.

System memory management functions consist of the extended SVCs for system memory operation that allocate and release memory from the system memory, and the memory allocation libraries that manage memory through subdividing system memory obtained in blocks into smaller ones.

The system memory management functions are for use not only within T-Kernel but also in applications, subsystems, and device drivers. Use inside T-Kernel does not have to go through extended SVC; this choice is implementation-dependent.

5.1.1 System Memory Allocation

System memory allocation functions provide extended SVCs for allocating and releasing memory from the system memory and referring to the system memory information.

5.1.1.1 tk_get_smb - Allocate System Memory

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_get_smb (void **addr , INT nblk , UINT attr);

Parameter

void**	addr	Pointer to Memory Start Address	Pointer to the area to return the start address of the allocated memory
INT	nblk	Number of Blocks	Number of memory blocks to be allocated
UINT	attr	Attribute	Attribute for memory to be allocated

Return Parameter

ER	ercd	Error Code	Error code
void*	addr	Memory Start Address	Start address of the allocated memory

Error Code

E_OK	Normal completion
E_PAR	Parameter error ((nblk≦0) or attr is invalid)
E_NOMEM	Insufficient memory (system memory is insufficient)
E_MACV	Memory access privilege error (unable to write to addr)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Allocates a contiguous memory area having the size of the number of memory blocks specified in nblk, and having the attributes specified in attr. The start address of the allocated memory space is returned in addr.

The following attributes can be specified in attr:

```
attr := (TA_RNG0 || TA_RNG1 || TA_RNG2 || TA_RNG3) | [TA_NORESIDENT]
```

TA_RNG0	Specify the protect level 0 memory
TA_RNG1	Specify the protect level 1 memory
TA_RNG2	Specify the protect level 2 memory
TA_RNG3	Specify the protect level 3 memory
TA_NORESIDENT	Specify nonresident memory

TA_RNGn is specified to limit the protection levels from which memory can be accessed. Only tasks running at the same or higher protection level than the one specified can access the allocated memory.

When TA_NORESIDENT is specified, the allocated memory becomes nonresident. In a system without MMU, the actual behavior is the same as the resident memory even if the nonresident memory attribute is specified, but an error is not returned.

If a negative value is specified in nblk or an unavailable attribute is specified in attr, the error code E_PAR is returned. When the write access to the memory (the area to return the start address of the allocated memory) pointed by addr is not allowed, the error code E_MACV is returned.

If the contiguous memory space for the number of blocks specified in nblk cannot be allocated, the error code E_NOMEM is returned. In this case, NULL is returned in the memory pointed by addr.

Additional Notes

In a system without MMU, the implementation cannot detect the access privilege error exception even if an access violates the memory protection level, which allows the access as normal. In consideration of program portability and expandability, it is recommended that the appropriate protection level for the protection levels of accessing tasks is specified for the memory to be allocated.

5.1.1.2 tk_rel_smb - Release System Memory

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_rel_smb (void *addr);

Parameter

void*	addr	Memory Start Address	Start address of memory to be released

Return Parameter

ER	ercd	Error Code	Error code

Error Code

E_OK	Normal completion
E_PAR	Parameter error (invalid addr)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Releases the resident memory specified in addr. addr must be the address retrieved by tk_get_smb().

If the address specified in addr is invalid, the error code E_PAR is returned. Specifically, when addr points at the space out of the memory range managed by T-Kernel or when the memory already released by tk_rel_smb() is released again, the error code E_PAR is returned. However, due to implementation constraints, an error may not be detected even if addr is invalid. In that case, the subsequent correct behavior is not guaranteed. The caller must guarantee the validity of addr.

5.1.1.3 tk_ref_smb - Reference System Memory Block

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_smb (T_RSMB *pk_rsmb);

Parameter

T_RSMB*	pk_rsmb	Packet to Return System Memory Block information	Pointer to the area to return the system memory information

Return Parameter

ER	ercd	Error Code	Error code

pk_rsmb Detail:

INT	blksz	Block Size	Block size (in bytes)
INT	total	Total Block Count	Total block count
INT	free	Free Block Count	Remaining free block count

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

Error Code

E_OK	Normal completion
E_MACV	Memory access privilege error (unable to write to pk_rsmb)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Gets information about system memory.

A system with the virtual memory can use a memory larger than the physical memory by allocating the non-resident memory. For this reason, total number of blocks or the number of remaining free blocks may not be uniquely determined. 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.

5.1.2 Memory Allocation Library Functions

Memory allocation library is used to efficiently use memory by subdividing system memory obtained in blocks by tk_get_smb() into smaller ones.

System memory returned by tk_get_smb() is managed inside the memory allocation library, and the memory of the size requested from an application is allocated from that memory. If the free memory managed by the memory allocation library is smaller than the one requested from an application, additional memory is allocated by calling tk_get_smb() again.

On the other hand, when memory is returned from an application, if the entire memory block containing the returned memory becomes free (unallocated), that memory block is released by calling tk_rel_smb(). The strict timing, however, of allocating or releasing memory block is implementation-dependent.

Memory allocation library provides functions equivalent to `malloc/calloc/realloc/free` provided by C standard library. If a target memory is nonresident memory, its API has a name beginning with the letter V, and if a target memory is resident memory, its API has a name beginning with the letter K.

These memories are all allocated as memory with a protection level specified in `TSVCLimit`.

5.1.2.1 Vmalloc - Allocate Nonresident Memory

C Language Interface

```
#include <tk/tkernel.h>
```

void* Vmalloc (size_t size);

Parameter

size_t	size	Size	Memory size to be allocated (in bytes)

Return Parameter

void*	addr	Memory Start Address	Start address of the allocated memory

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Allocates the nonresident memory `size` bytes and returns the start address of the allocated memory in `addr`.

When the specified size of memory cannot be allocated or 0 is specified in `size`, `NULL` is returned in `addr`.

APIs in the memory allocation library, including Vmalloc, cannot be called from a task-independent portion and while dispatch or interrupt is disabled. Such a call may lead to an undefined behavior including possible system failure, and the caller is responsible for guaranteeing the state on the call.

Additional Notes

Any value can be specified in `size`. Note that a larger memory size than the number of bytes specified in `size` may be allocated internally for allocating the management space, aligning the allocated memory address, or other reasons. For example, when the implementation specifies that the least allocatable memory size is 16 bytes and the alignment is 8-byte unit, 16-byte memory is allocated internally even if a value less than 16 bytes is specified in `size`. Similarly, 24-byte memory is allocated even if 20 bytes is specified in `size`.

Therefore, when comparing the entire system memory size used by the memory allocation library with the total memory size allocated by individual APIs in the memory allocation library, the former value may be larger.

5.1.2.2 Vcalloc - Allocate Nonresident Memory

C Language Interface

```
#include <tk/tkernel.h>
```

void* Vcalloc (size_t nmemb , size_t size);

Parameter

size_t	nmemb	Number of Memory Blocks	Number of memory blocks to be allocated
size_t	size	Size	Memory block size to be allocated (in bytes)

Return Parameter

void*	addr	Memory Start Address	Start address of the allocated memory

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Allocates the specified number (nmemb) of contiguous memory blocks of the specified bytes (size), clears them with 0, then returns the start address of them in addr. This memory allocation operation is identical to allocating one memory block of the number of size * nmemb bytes. The allocated memory is nonresident memory.

When the specified number of memory blocks cannot be allocated or 0 is specified in nmemb or size, NULL is returned in addr.

APIs in the memory allocation library, including Vcalloc, cannot be called from a task-independent portion and while dispatch or interrupt is disabled. Such a call may lead to an undefined behavior including possible system failure, and the caller is responsible for guaranteeing the state on the call.

Additional Notes

A larger memory size than the number of size * nmemb bytes may be allocated internally. For more details, see the additional note for Vmalloc().

5.1.2.3 Vrealloc - Reallocate Nonresident Memory

C Language Interface

```
#include <tk/tkernel.h>
```

void* Vrealloc (void *ptr , size_t size);

Parameter

void*	ptr	Pointer to Memory	Memory address to be reallocated
size_t	size	Size	Reallocated memory size (in bytes)

Return Parameter

void*	addr	Memory Start Address	Start address of the reallocated memory

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Changes the size of the previously allocated nonresident memory specified in ptr to the size specified in size. At that time, reallocates the memory and returns the start address of the reallocated memory in addr.

Generally, addr results in different value from ptr because the memory start address is moved by reallocating the memory with resizing. The content of the reallocated memory is retained. To do so, the memory content is copied during the Vrealloc processing. The memory that becomes free by reallocation will be released.

The start address of the memory allocated previously by Vmalloc, Vcalloc, or Vrealloc must be specified in ptr. The caller must guarantee the validity of ptr.

If NULL is specified in ptr, only the new memory allocation is performed. This operation is identical to Vmalloc().

When the specified size of memory cannot be reallocated or 0 is specified in size, NULL is returned in addr. In this case, the memory specified by ptr is only released if a value other than NULL is specified in ptr. This operation is identical to Vfree().

APIs in the memory allocation library, including Vrealloc, cannot be called from a task-independent portion and while dispatch or interrupt is disabled. Such a call may lead to an undefined behavior including possible system failure, and the caller is responsible for guaranteeing the state on the call.

Additional Notes

The memory address returned in addr may be the same as ptr in some cases, for example, when the memory size becomes smaller than before by reallocation or when the reallocation is performed without moving the memory start address because an unallocated memory area was around the memory specified in ptr.

A larger memory size than the number of bytes specified in `size` may be allocated internally. For more details, see the additional note for Vmalloc().

5.1.2.4 Vfree - Release Nonresident Memory

C Language Interface

```
#include <tk/tkernel.h>

void Vfree (void *ptr );
```

Parameter

void*	ptr	Pointer to Memory	Start address of memory to be released

Return Parameter

None

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Releases the nonresident memory specified in `ptr`.

The start address of the memory allocated previously by Vmalloc, Vcalloc, or Vrealloc must be specified in `ptr`. The caller must guarantee the validity of `ptr`.

APIs in the memory allocation libraries, including Vfree, cannot be called from a task-independent portion and while dispatch or interrupt is disabled. Such a call may lead to an undefined behavior including possible system failure, and the caller is responsible for guaranteeing the state on the call.

5.1.2.5 Kmalloc - Allocate Resident Memory

C Language Interface

```
#include <tk/tkernel.h>
```

void* Kmalloc (size_t size);

Parameter

size_t	size	Size	Memory size to be allocated (in bytes)

Return Parameter

void*	addr	Memory Start Address	Start address of the allocated memory

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Allocates the resident memory of bytes specified in `size` and returns the start address of the allocated memory in `addr`.

When the specified size of memory cannot be allocated or 0 is specified in `size`, `NULL` is returned in `addr`.

APIs in the memory allocation library, including Kmalloc, cannot be called from a task-independent portion and while dispatch or interrupt is disabled. Such a call may lead to an undefined behavior including possible system failure, and the caller is responsible for guaranteeing the state on the call.

Additional Notes

A larger memory size than the number of bytes specified in `size` may be allocated internally. For more details, see the additional note for Vmalloc().

5.1.2.6 Kcalloc - Allocate Resident Memory

C Language Interface

```
#include <tk/tkernel.h>
```

void* Kcalloc (size_t nmemb , size_t size);

Parameter

size_t	nmemb	Number of Memory Blocks	Number of memory blocks to be allocated
size_t	size	Size	Memory block size to be allocated (in bytes)

Return Parameter

void*	addr	Memory Start Address	Start address of the allocated memory

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Allocates the specified number (nmemb) of contiguous memory blocks of the specified bytes (size), clears them with 0, then returns the start address of them in addr. This memory allocation operation is identical to allocating one memory block of the number of size * nmemb bytes. The allocated memory is a resident memory.

When the specified number of memory blocks cannot be allocated or 0 is specified in nmemb or size, NULL is returned in addr.

APIs in the memory allocation libraries, including Kcalloc, cannot be called from a task-independent portion and while dispatch or interrupt is disabled. Such a call may lead to an undefined behavior including possible system failure, and the caller is responsible for guaranteeing the state on the call.

Additional Notes

A larger memory size than the number of size * nmemb bytes may be allocated internally. For more details, see the additional note for Vmalloc().

5.1.2.7 Krealloc - Reallocate Resident Memory

C Language Interface

```
#include <tk/tkernel.h>
```

void* Krealloc (void *ptr , size_t size);

Parameter

void*	ptr	Pointer to Memory	Memory address to be reallocated
size_t	size	Size	Reallocated memory size (in bytes)

Return Parameter

void*	addr	Memory Start Address	Start address of the reallocated memory

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Changes the size of the previously allocated resident memory specified in ptr to the size specified in size. At that time, reallocates the memory and returns the start address of the reallocated memory in addr.

Generally, addr results in different value from ptr because the memory start address is moved by reallocating the memory with resizing. The content of the reallocated memory is retained. To do so, the memory content is copied during the Krealloc processing. The memory that becomes free by reallocation will be released.

The start address of the memory allocated previously by Kmalloc, Kcalloc, or Krealloc must be specified in ptr. The caller must guarantee the validity of ptr.

If NULL is specified in ptr, only the new memory allocation is performed. This operation is identical to Kmalloc().

When the specified size of memory cannot be reallocated or 0 is specified in size, NULL is returned in addr. In this case, the memory specified by ptr is only released if a value other than NULL is specified in ptr. This operation is identical to Kfree().

APIs in the memory allocation library, including Krealloc, cannot be called from a task-independent portion and while dispatch or interrupt is disabled. Such a call may lead to an undefined behavior including possible system failure, and the caller is responsible for guaranteeing the state on the call.

Additional Notes

The memory address returned in addr may be the same as ptr in some cases, for example, when the memory size becomes smaller than before by reallocation or when the reallocation is performed without moving the memory start address because an unallocated memory area was around the memory specified in ptr.

A larger memory size than the number of bytes specified in `size` may be allocated internally. For more details, see the additional note for Vmalloc().

5.1.2.8 Kfree - Release Resident Memory

C Language Interface

```
#include <tk/tkernel.h>
```

void Kfree (void *ptr);

Parameter

void*	ptr	Pointer to Memory	Start address of memory to be released

Return Parameter

None

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Releases the resident memory specified in ptr.

The start address of the memory allocated previously by Kmalloc, Kcalloc, or Krealloc must be specified in ptr. The caller must guarantee the validity of ptr.

APIs in the memory allocation library, including Kfree, cannot be called from a task-independent portion and while dispatch or interrupt is disabled. Such a call may lead to an undefined behavior including possible system failure, and the caller is responsible for guaranteeing the state on the call.

5.2 Address Space Management Functions

Address space management functions perform various operations or managements on logical address spaces. These functions are mainly realized by operating on MMUs or page tables, and offer address space configuration functions to set a task space, address space checking that checks access privilege, locking memory space (resident), and conversion and mapping between logical address and physical address.

These functions are used to not only implement system programs such as device drivers and subsystems but also to realize a virtual memory system by combining subsystems that process a demand paging related processing.

APIs for address space management functions are provided even for systems that do not use MMUs. In consideration of portability and expandability, it is preferable for applications to use these APIs appropriately even on the systems that do not use MMUs.

In T-Kernel, four levels from 0 to 3 (meaning privileged mode, user mode, etc.) are defined as the protection level at runtime, and also four levels from 0 to 3 are defined as the protection level of memory to be accessed. The currently running execution task can access only to the memory with the same or lower protection level. An MMU is responsible for checking the memory privileges at runtime. This function is useful for protecting a system such as OS from being illegally accessed by programs. To realize a memory access privileges check function, T-Kernel sets configuration of MMU and others appropriately.

Caller access privilege information of memory is held for each task to indicate the access right of a protection level immediately before an extended SVC is called. As the information indicates the protection level prior to an extended SVC, it may not be identical to the current protection level at runtime. For example, when a task that is running at protection level 3 calls an extended SVC that usually runs at protection level 0, the task will have an access right of protection level 3. When extended SVC (a) calls extended SVC (b), making a nested call, the caller access privilege information at the extended SVC (b) that is called in a nested manner has a protection level of immediately before the extended SVC (b) has been called, which means the protection level 0 under which the extended SVC (a) is running.

Caller access privilege information of memory is set as follows.

- Immediately after a task is started, the protection level at runtime specified when the task was created is set as the caller access privilege information.
- When an extended SVC is called, the protection level at the time of the call is set as the caller access privilege information.
- Upon return from the extended SVC, the caller access privilege information reverts to that at the time the extended SVC was called.
- When SetTaskSpace() is issued, the protection level of the specified task just before the call to an extended SVC is set as the caller access privilege information of the invoking task. When the call to extended SVC is nested, the protection level at runtime just before the last call to the extended SVC is set. When the specified task is running a task portion, the protection level at runtime specified when the task was created is set as the caller access privilege information of the running task.

Caller access privilege information of memory is maintained in order for extended SVC to support operations depending on the protection levels of callers. For example, address space checking functions (ChkSpaceXXX) can be used in extended SVC to check memory access privilege of the caller, as it utilizes caller access privilege information instead of current protection level at runtime.

5.2.1 Address Space Configuration

How to handle T-Kernel address space is explained in Section 2.7.1, “Address Space”. Address space configuration functions provide APIs for setting task address spaces and caller access privilege information.

5.2.1.1 SetTaskSpace - Set Task Space

C Language Interface

```
#include <tk/tkernel.h>

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

Parameter

ID	tskid	Task ID	Task ID of the task which has the source address space

Return Parameter

ER	ercd	Error Code	Error code

Error Code

E_OK	Normal completion
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

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Sets the task space and caller access privilege information of the invoking task according to the task specified in tskid. As a result, the task that executed this API has the same address space as the task with tskid, and the protection level of the specified task just before the call to an extended SVC is set as the caller access privilege information of the invoking task. When the call to extended SVC is nested, the protection level at runtime just before the last call to the extended SVC is set. When the specified task is running a task portion, the protection level at runtime specified when the specified task was created is set as the caller access privilege information of the invoking task.

Note that, even if the address space or caller access privilege information of the task tskid (target task) is changed after executing this API, the address space or caller access privilege information of the invoking task is not affected. This means that only the state of the target task at the time when executing this API is reflected to that of the invoking task. The invoking task does not follow the later states of the target task.

When this API is executed during an extended SVC and the extended SVC returns to the caller, its caller access privilege information is restored to the state prior to calling the extended SVC. However, its task space is not restored. The task space set by this API is still valid after the extended SVC returned to the caller.

The task ID of the invoking task cannot be specified in tskid. If TSK_SELF is used to specify the invoking task, caller access privilege information is set to the currently running protection level; task space is not switched in this case.

Note that the protection level at runtime is not altered after changing caller access privilege information.

Additional Notes

In the situation that a task A (a task that calls an extended SVC for the device management or subsystem) requests another task B to manage a device driver or subsystem, SetTaskSpace() is used to set the task space and caller access privilege information of the managing task B as the same as those of the requestor task A.

For example, it is assumed that the managing task B for the device driver reads the input data from the device and stores it in the buffer X specified by the requestor task A. If the address of the buffer X is included in the task space of the task A, and the requestor task A and the managing task B have different task spaces, the managing task B cannot access the buffer X and store the input data in it.

In such a case, the managing task B can execute SetTaskSpace() in advance to set its task space as the same as that of the requestor task A to access the buffer X. Since the caller access privilege information of the managing task B becomes the same as that of the requestor task A, it is checked appropriately when storing the input data in the buffer X.

Use tk_set_tsp to set the task space only without setting the caller access privilege information.

5.2.2 Address Space Checking

The following functions check whether access is allowed to the specified memory space, based on the current caller access privilege information.

APIs named ChkSpaceXXX() are provided for such checking. The last letter of API name means as follows:

- R: Check for read access privilege.
- RW: Check for read and write access privilege.
- RE: Check for read and execute access privilege.

If the current caller access privilege information does not allow access to the target memory space, or memory does not exist in the target memory space, an error code E_MACV is returned. The same error code E_MACV is returned when the access is not allowed for a part of the target memory space, or a part of memory does not exist in the target memory space.

If the target memory space for checking is a task space, the currently set task space is used.

Additional Notes
When a general application task A running at lower protection level requests a device driver or subsystem running at higher protection level for a processing, if a parameter or return parameter of the processing is placed in the memory space X, a check should be performed by the device driver or subsystem side to check if the requesting task A has access privileges for the memory space X. If this check is not performed, task A can, for example, easily access the disallowed memory space via a device driver or subsystem illegally. APIs for address space checking are functions that are assumed to be used to perform such a check in these situation.

5.2.2.1 ChkSpaceR - Check Read Access Privilege

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = ChkSpaceR (CONST void *addr , INT len);

Parameter

CONST void*	addr	Memory Start Address	Start address of the target memory
INT	len	Length	Size of the target memory (in bytes)

Return Parameter

ER	ercd	Error Code	Error code

Error Code

E_OK	Normal completion
E_MACV	Memory cannot be accessed

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Based on the current caller access privilege information, checks whether read access is allowed to the `len` bytes memory area from the address specified in `addr`. E_OK is returned if access is allowed; E_MACV is returned otherwise.

5.2.2.2 ChkSpaceRW - Check Read-Write Access Privilege

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = ChkSpaceRW (CONST void *addr , INT len);

Parameter

CONST void*	addr	Memory Start Address	Start address of the target memory
INT	len	Length	Size of the target memory (in bytes)

Return Parameter

ER	ercd	Error Code	Error code

Error Code

E_OK	Normal completion
E_MACV	Memory cannot be accessed

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Based on the current caller access privilege information, checks whether both read and write accesses are allowed to the `len` bytes memory area from the address specified in `addr`. E_OK is returned if both accesses are allowed; E_MACV is returned if at least one is prohibited.

5.2.2.3 ChkSpaceRE - Check Read-Execute Access Privilege

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = ChkSpaceRE (CONST void *addr , INT len);

Parameter

CONST void*	addr	Memory Start Address	Start address of the target memory
INT	len	Length	Size of the target memory (in bytes)

Return Parameter

ER	ercd	Error Code	Error code

Error Code

E_OK	Normal completion
E_MACV	Memory cannot be accessed

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Based on the current caller access privilege information, checks whether read access is allowed to the `len` bytes memory area from the address specified in `addr` and whether that memory area can be executed as a program. E_OK is returned if both are allowed; E_MACV is returned if at least one is prohibited.

5.2.2.4 ChkSpaceBstrR - Check Read Access Privilege (String)

C Language Interface

```
#include <tk/tkernel.h>
```

INT rlen = ChkSpaceBstrR (CONST UB *str , INT max);

Parameter

CONST UB*	str	String	Start address of the target string
INT	max	Max Length	Maximum length of the target string

Return Parameter

INT	rlen		Result Length	Length of the accessible string (in bytes)
		or	Error Code	Error code

Error Code

E_MACV	Memory cannot be accessed

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Based on the current caller access privilege information, checks whether read access and write access is allowed to the memory area from str to the string termination ('¥0') or to the number of characters (bytes) specified in max, whichever comes first. If max = 0 is set, 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 occurs before max bytes, the length to the character before '¥0' is returned; if max characters are scanned before the string termination is seen, max is returned.

If access is prohibited, the error code E_MACV is returned.

5.2.2.5 ChkSpaceBstrRW - Check Read-Write Access Privilege (String)

C Language Interface

```
#include <tk/tkernel.h>
```

INT rlen = ChkSpaceBstrRW (CONST UB *str , INT max);

Parameter

CONST UB*	str	String	Start address of the target string
INT	max	Max Length	Maximum length of the target string

Return Parameter

INT	rlen		Result Length	Length of the accessible string (in bytes)
		or	Error Code	Error code

Error Code

E_MACV	Memory cannot be accessed

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Based on the current caller access privilege information, checks whether read access and write access is allowed to the memory area from str to the string termination ('¥0') or to the number of characters (bytes) specified in max, whichever comes first. If max = 0 is set, privilege is checked up to the string termination.

If both read and write access is allowed, the length of the string (bytes) is returned. If the string termination occurs before max bytes, the length to the character before '¥0' is returned; if max characters are scanned before the string termination is seen, max is returned.

If at least one of read and write accesses is prohibited, the error code E_MACV is returned.

5.2.2.6 ChkSpaceTstrR - Check Read Access Privilege (TRON Code)

C Language Interface

```
#include <tk/tkernel.h>
```

INT rlen = ChkSpaceTstrR (CONST TC *str , INT max);

Parameter

CONST TC*	str	String	Start address of the target string
INT	max	Max Length	Maximum length of the target string

Return Parameter

INT	rlen		Result Length	Length of the accessible string (in TRON code characters)
		or	Error Code	Error code

Error Code

E_MACV	Memory cannot be accessed

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Based on the current caller access privilege information, checks whether read access is allowed to the memory area from `str` to the TRON code string termination (`TNULL` = 0x0000) or to the number of characters (number of TRON code characters) specified in `max`, whichever comes first. If `max` = 0 is set, privilege is checked up to the string termination.

If access is allowed, the length of the string (number of TRON code characters) is returned. If the string termination occurs before `max` TRON code characters, the length to the character before `TNULL` is returned; if `max` characters are scanned before the string termination is seen, `max` is returned.

If access is prohibited, the error code E_MACV is returned.

`str` must be an even-numbered address.

5.2.2.7 ChkSpaceTstrRW - Check Read-Write Access Privilege (TRON Code)

C Language Interface

```
#include <tk/tkernel.h>
```

INT rlen = ChkSpaceTstrRW (CONST TC *str , INT max);

Parameter

CONST TC*	str		String	Start address of the target string
INT	max		Max Length	Maximum length of the target string (in TRON code characters)

Return Parameter

INT	rlen		Result Length	Length of the accessible string
		or	Error Code	Error code

Error Code

E_MACV	Memory cannot be accessed

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Based on the current caller access privilege information, checks whether read access is and write access allowed to the memory area from `str` to the TRON code string termination (`TNULL` = 0x0000) or to the number of characters (number of TRON code characters) specified in `max`, whichever comes first. If `max` = 0 is set, privilege is checked up to the string termination.

If both read and write access is allowed, the length of the string (number of TRON code characters) is returned. If the string termination occurs before `max` TRON code characters, the length to the character before `TNULL` is returned; if `max` characters are scanned before the string termination is seen, `max` is returned.

If at least one of read and write accesses is prohibited, the error code E_MACV is returned.

`str` must be an even-numbered address.

5.2.3 Logical Address Space Management

Logical address space management functions provide APIs relating to converting address (conversion from a logical address to a physical address), making memory resident, and setting memory access privileges.

T-Kernel performs the address conversion (conversion from a logical address to a physical address) using MMUs in order to manage the access privileges for memory, realize a task space, and use memory efficiently. While usual programs does not need to handle any physical address because they are running in a logical address space, some system programs that directly operate a hardware device such as a device driver that performs DMA transfer may handle physical addresses. Since mapping between a logical address and a physical address must be retrieved or set, CnvPhysicalAddr(), MapMemory(), and UnmapMemory() are provided as APIs for these operations.

When a virtual memory system is constructed on T-Kernel, a situation occurs where memory being accessed from program A does not physically exist in main memory (paged out state). When there is an access to the paged-out memory, in the case of some CPU, an MMU detects the access and raises a page fault CPU exception, letting the virtual memory system that processes the exception returns (pages in) the paged out memory content from an external disk (secondary storage device) to memory. As this processing is performed, program A can proceed with its processing regardless of whether the accessed memory is paged out or not. This is the general implementation method of a virtual memory system.

The above page-in processing cannot be performed when a page fault occurs, however, for those programs which are during execution of task-independent portion, dispatch disabled, or interrupts disabled. For this reason, to avoid page faults during execution of a program, all the memory to be accessed must be made resident in advance by paging them in. The same action also needs to be performed when performing a DMA transfer or executing a strictly time-constrained program. LockSpace() for locking (making resident) memory space and UnlockSpace() for releasing the locked memory space are provided as APIs for this kind of situation.

In addition to this, GetSpaceInfo() to retrieve various information on address space, SetMemoryAccess() to set memory access privileges, etc. are provided as APIs.

APIs that perform processing related to DMA transfer also perform memory cache control optimized to the DMA transfer. Specifically, when performing a conversion from a logical address to a physical address using CnvPhysicalAddr(), memory caching for the target memory space is turned off so that DMA transfer can be performed. After completing the DMA transfer, making memory nonresident by executing UnlockSpace() returns memory caching to the on state.

Additional Notes

T-Kernel/SM sets or operates MMUs and page tables in order to manage correspondence relationship (mapping) between a logical space and a physical space, memory access privileges, page nonexistence, making a page resident, etc. However, T-Kernel is not the sole entity to realize a virtual memory system. In order to actually realize a virtual memory system, other various processings such as page in/out between the physical memory and the disk (secondary storage device) are required. These processings are performed by subsystems (part of T-Kernel Extension) for realizing a virtual memory system rather than T-Kernel itself.

5.2.3.1 LockSpace - Lock Memory Space

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = LockSpace (CONST void *addr , INT len);

Parameter

CONST void*	addr	Memory Start Address	Start address of memory to be locked
INT	len	Length	Size of memory to be locked (in bytes)

Return Parameter

ER	ercd	Error Code	Error code

Error Code

E_OK	Normal completion
E_PAR	Parameter error (len ≦ 0)
E_MACV	An area out of the memory space is specified
E_NOMEM	Insufficient memory (page in memory for resident cannot be allocated)
E_LIMIT	Lock attempts exceed the upper limit of the number of locks

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Locks (makes resident) the len bytes memory area from the logical address addr (target area). After making an area resident with this API, the target area will not be paged out, and is always mapped to a physical address space and allocated to the real memory (physical memory).

When a part of the target area has been paged out, it is paged in before making the area resident. If the real memory cannot be allocated for paging in, the error code E_NOMEM is returned.

LockSpace() can be executed more than once for the same memory area. In this case, the number of LockSpace() operations is counted and the same number of UnlockSpace() operations can make the area nonresident. That is, the resident state can be nested by LockSpace(). However, there is an implementation-dependent upper limit to the nesting depth (difference between the numbers of LockSpace() and UnlockSpace() operations). If LockSpace() is executed exceeding the upper limit, the error code E_LIMIT is returned.

If 0 or less is specified in len, the error code E_PAR is returned. If the target area includes an area out of the memory space (logical address that is not assumed to be allocated to memory), the error code E_MACV is returned.

The lock operation (making resident) with this API is performed in units of page, using the MMU function. Therefore, if addr is not the start address of a page or len is not an integral multiple of the page size, the entire

pages containing the range specified by `addr` and `len` are taken as the target area. For example, if 1 is specified in `len`, one page area is locked.

In a system without MMU, all the memory can be considered resident. Thus, no specific operation must be performed in LockSpace(), but E_OK must be returned rather than an error code, in consideration of compatibility with a system using MMU. In a system without MMU, whether or not to check errors such as E_PAR is implementation-dependent.

Additional Notes

Among memory resident operations with LockSpace(), the page-in and some other operations are performed by calling the subsystem to realize a virtual memory system. The calling interface is implementation-dependent.

An area in logical address space allocated by MapMemory() must not be included in the target area for LockSpace(). The subsequent correct behavior of the whole system in such a case is not guaranteed.

5.2.3.2 UnlockSpace - Unlock Memory Space

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = UnlockSpace (CONST void *addr , INT len);

Parameter

CONST void*	addr	Memory Start Address	Start address of memory to be unlocked
INT	len	Length	Size of memory to be unlocked (in bytes)

Return Parameter

ER	ercd	Error Code	Error code

Error Code

E_OK	Normal completion
E_PAR	Parameter error (len ≦ 0)
E_MACV	An area out of the memory space is specified
E_LIMIT	Non-locked area was specified

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Unlocks (makes nonresident) the len bytes area from the logical address addr (target area). After being made nonresident with this API, the target area will be subject to page-out.

If the memory cache mode was off for the target area, it is turned on.

The target area must be the same area that was specified when LockSpace() was issued to lock it. Note that it is not possible to unlock just a part of a locked area. In addition, T-Kernel cannot detect such an operation as an error. The caller is responsible for specifying the same area.

When LockSpace() was executed more than once for the same memory area, the same number of UnlockSpace() operations can make it nonresident. UnlockSpace() returns E_OK rather than an error even if the memory area is not made nonresident because the number of UnlockSpace() operations does not reach the number of LockSpace() operations. On the other hand, if a non-locked area is specified for the target area, E_LIMIT is returned as the lock count error.

If 0 or less is specified in len, the error code E_PAR is returned. If the target area includes an area out of the memory space (logical address that is not assumed to be allocated to memory), the error code E_MACV is returned.

The unlock operation (making nonresident) with this API is performed in units of page, using the MMU function. Therefore, if addr is not the start address of a page or len is not an integral multiple of the page size, the

entire pages containing the range specified by `addr` and `len` are taken as the target area. For example, if 1 is specified in `len`, one page area is unlocked.

In a system without MMU, all the memory can be considered resident. Thus, no specific operation must be performed in UnlockSpace() as well as LockSpace(), but E_OK must be returned rather than an error code, in consideration of compatibility with a system using MMU. In a system without MMU, whether or not to check errors such as E_PAR is implementation-dependent.

Additional Notes

A logical address area allocated by MapMemory() must not be included in the target area for UnlockSpace(). The subsequent correct behavior of the whole system is not guaranteed in such a case.

When performing the DMA transfer, the buffer memory area must be made resident and the buffer physical address must be set on the DMA controller after turning off the memory cache mode setting. Normal steps are as follows:

1. Use LockSpace() to make the buffer resident.
2. Use CnvPhysicalAddr() to get the buffer physical address and turn off the buffer memory cache mode setting.
3. Perform the DMA transfer between the buffer and the I/O device.
4. Use UnlockSpace() to make the buffer nonresident and turn on the buffer memory cache mode setting.

UnlockSpace() always turns on the memory cache mode setting regardless of previously issued APIs as shown above. Note that the memory cache mode setting may be changed by executing UnlockSpace().

5.2.3.3 CnvPhysicalAddr - Get Physical Address

C Language Interface

```
#include <tk/tkernel.h>
```

INT rlen = CnvPhysicalAddr (CONST void *vaddr , INT len , void **paddr);

Parameter

CONST void*	vaddr	Virtual Address	Logical address of the source
INT	len	Length	Memory area size (in bytes)
void**	paddr	Pointer to Physical Address	Pointer to the area to return the physical address corresponding to the logical address

Return Parameter

INT	rlen		Result Length	Size of contiguous physical address area (in bytes)
		or	Error Code	Error code
void*	paddr		Physical Address	Physical address corresponding to the logical address

Error Code

E_OK	Normal completion
E_PAR	Parameter error (cache of the target area cannot be controlled)
E_MACV	An area out of the memory space is specified

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Gets the physical address corresponding to the logical address `vaddr`, returning the result in `paddr`. Also returns the contiguous size (in bytes) of the corresponding physical address in the return code `rlen`, within the `len` bytes memory area from `vaddr`. That is, the contiguous correspondence between the logical address and the physical address exists only for `rlen` in size (`rlen`≦`len`). The contiguous area in the logical address space from `vaddr` for `rlen` corresponds to the contiguous area in the physical address space from `paddr` for `rlen`.

Also turns off the memory cache mode setting for the physical address area from `paddr` for `rlen` (target area). This assumes that the DMA transfer is performed after executing CnvPhysicalAddr(). If it is not possible to make memory cached off partly by a hardware limitation, this API flush the cache memory (that is, write back it and invalidate it).

CnvPhysicalAddr() does not make the target area resident. Before performing the DMA transfer, the buffer area must be made resident (locked) by separately issuing LockSpace() for the buffer area.

If 0 or less is specified in `len`, the error code E_PAR is returned. If the `len` bytes memory area from `vaddr`

includes an area out of the memory space (logical address that is not assumed to be allocated to memory), the error code E_MACV is returned.

Additional Notes

The CnvPhysicalAddr() API is intended to be used for preparing the DMA transfer. For concrete usage for the DMA transfer, see the additional note for UnlockSpace().

For the target area of CnvPhysicalAddr(), it is best to set the memory attribute that guarantees the completion of memory access in addition to turning off the cache mode setting.

5.2.3.4 MapMemory - Map Memory

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = MapMemory (CONST void *paddr , INT len , UINT attr , void **laddr);

Parameter

CONST void*	paddr	Physical Address	Physical address to be mapped
INT	len	Length	Size of memory to be mapped (in bytes)
UINT	attr	Attribute	Memory attribute for mapping
void**	laddr	Pointer to Logical Address	Pointer to the area to return the mapped logical address

Return Parameter

ER	ercd	Error Code	Error code
void*	laddr	Logical Address	Mapped logical address

Error Code

E_OK	Normal completion
E_PAR	Parameter error (len ≦ 0)
E_LIMIT	Insufficient logical address space to be mapped
E_NOMEM	Insufficient real memory for allocating or insufficient memory for managing logical address space

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Maps the len bytes contiguous area from the physical address paddr to a logical address space and returns the mapped logical start address in laddr. The mapped memory area is made resident (locked). The attributes specified in attr are set for the mapped memory area.

The following attributes can be specified in 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

Other attributes may be specified depending on the hardware or implementation.

If NULL is specified in paddr, the actual contiguous len bytes physical memory is allocated for address and the real memory physical address space is mapped to the logical address space.

If 0 or less is specified in len, the error code E_PAR is returned. If the allocation fails due to insufficient mapped logical address space, the error code E_LIMIT is returned. If the memory required to manage the logical address space cannot be allocated or the real memory cannot be allocated when NULL is specified in paddr, the error code E_NOMEM is returned.

Additional Notes

MapMemory() has the function to map the space for an I/O device (Video RAM etc.) located in the physical address space to the logical address space that can be accessed directly from a program such as a device driver.

The mapped logical address laddr is automatically allocated during execution of this API. The mapped logical address cannot be specified.

An address within the system memory managed by T-Kernel cannot be specified in paddr. When you want to reserve the system memory with MapMemory(), specify NULL in paddr to use the system memory that is automatically allocated by T-Kernel.

Values corresponding to symbols (mnemonics) for attributes specified in attr may vary depending on implementation. Therefore, the above symbols should be used for attr, in consideration of compatibility.

MapMemory() must not be executed for the physical address area that is already a target of MapMemory(). The memory allocated by MapMemory() is a resident memory and cannot be made nonresident, so UnlockSpace() should not be called to make it nonresident. The caller is responsible for preventing such usage.

After executing MapMemory(), directly accessing paddr or subsequent physical address using other method rather than via the logical address allocated as laddr may cause cache inconsistency or other problem. With such an access, the caller is responsible for paying careful attention to data consistency.

If MM_CDIS is specified for attr, the memory attribute should guarantee the completion of memory access in addition to not using cache.

5.2.3.5 UnmapMemory - Unmap Memory

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = UnmapMemory (CONST void *laddr);

Parameter

CONST void*	laddr	Logical Address	Logical address to be unmapped

Return Parameter

ER	ercd	Error Code	Error code

Error Code

E_OK	Normal completion
E_PAR	Parameter error (invalid laddr)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Unmaps (releases) a logical address area allocated by MapMemory(). The logical address of the area to be unmapped is specified in laddr. It must be the value retrieved from the return parameter laddr of MapMemory().

If paddr = NULL is specified to allocate the real memory when executing MapMemory(), it is also released by executing UnmapMemory().

5.2.3.6 GetSpaceInfo - Get Various Information about Address Space

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = GetSpaceInfo (CONST void *addr , INT len , T_SPINFO *pk_spinfo);

Parameter

CONST void*	addr	Start Address	Start logical address to get the information for
INT	len	Length	Space size to get the information for (in bytes)
T_SPINFO*	pk_spinfo	Packet to Return Address Space Info	Pointer to the area to return the address space information

Return Parameter

ER	ercd	Error Code	Error code

pk_spinfo Detail:

void*	paddr	Physical Address	Physical address corresponding to addr
void*	page	Page Start Address	Start physical address of the page that addr belongs to
INT	pagesz	Page Size	Page size (in bytes)
INT	cachesz	Cache Line Size	Cache line size (in bytes)
INT	cont	Continuous Length	Contiguous physical address space size (in bytes)

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

Error Code

E_OK	Normal completion
E_PAR	Parameter error (addr, len, or pk_spinfo is invalid or cannot be used)
E_MACV	Memory cannot be accessed; memory access privilege error

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Gets the address space information about the len bytes space from the logical address addr and returns it in the return parameter pk_spinfo. Returns the physical address corresponding to addr in paddr. Returns the start physical address of the page that addr belongs to in page.

Returns the page size in pagesz. The page size is defined in MMU and the same value as is used as unit for setting the memory access permission in SetMemoryAccess() or the cache mode in SetCacheMode().

Returns the cache line size in `cachesz`. The cache line size is the same value as is used as unit for controlling the cache in ControlCache().

Returns the contiguous size (in bytes) of the corresponding physical address in `cont`, within the `len` bytes space from `addr`. That is, the contiguous correspondence between the logical address and the physical address exists only for `cont` in size (`cont`≦`len`). The contiguous area in the logical address space from `addr` for `cont` corresponds to the contiguous area in the physical address space from `paddr` for `cont`.

If a paged out area exists in the range, physical addresses up to just before it are considered contiguous. Particularly, if a page to which `addr` belongs is paged out, `cont` = 0 is returned. In this case, E_OK is returned in the return code `ercd` and the contents other than `cont` in the return parameters of `pk_spinfo` are undefined.

If 0 or less is specified in `len`, the error code E_PAR is returned. When an error occurs, the contents set in `pk_spinfo` are undefined.

Difference from T-Kernel 1.0

This API was added in T-Kernel 2.0.

5.2.3.7 SetMemoryAccess - Set Memory Access Privilege

C Language Interface

```
#include <tk/tkernel.h>
```

INT rlen = SetMemoryAccess (CONST void *addr , INT len , UINT mode);

Parameter

CONST void*	addr	Start Address	Start logical address of the memory area to set the access permission for
INT	len	Length	Size of the memory area to set the access permission for (in bytes)
UINT	mode	Memory Access Mode	Mode indicating the memory access permission to be set

Return Parameter

INT	rlen		Result Length	Size of the area for which the memory access permission could be set (in bytes)
		or	Error Code	Error code

Error Code

E_OK	Normal completion
E_PAR	Parameter error (addr, len, or mode is invalid or cannot be used)
E_NOSPT	Unsupported function (function specified in mode is unsupported)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Sets the memory access permission specified in mode for the len bytes memory area from the logical address addr. Returns the size (in bytes) of the area for which the memory access permission can actually be set, in the return code rlen.

The following memory access permissions are specified in mode:

```
mode := ( MM_EXECUTE | MM_READ | MM_WRITE )
                MM_EXECUTE      Execution access
                MM_READ         Read access
                MM_WRITE        Write access
                    ...
                /* Implementation-dependent mode may be added */
```

Setting the memory access permission with this API is performed in units of page, using the MMU function. Therefore, if addr is not the start address of a page or len is not an integral multiple of the page size, the entire

pages containing the range specified by `addr` and `len` are taken as the target area for setting the memory access permission. For example, if 1 is specified in `len`, the memory access permission for one page is set.

Other memory access permissions may be specified depending on the hardware or implementation. Some or all of the above memory access permissions may not be set depending on the hardware or implementation. If any unavailable memory access permission is specified in `mode`, the error code E_NOSPT is returned.

Additional Notes

For the memory area used by normal applications, appropriate memory access permissions are set in advance by T-Kernel. Therefore, normal applications do not need to use SetMemoryAccess(). SetMemoryAccess() is intended for use by special-purpose programs rather than normal applications, for example, to allocate the system memory, dynamically manage the security, or debug programs.

The memory access permissions specified in `mode` are the same as some attributes specified in `attr` of MapMemory.

Difference from T-Kernel 1.0

This API was added in T-Kernel 2.0.

5.3 Device Management Functions

Device management functions manage device drivers running on T-Kernel.

A device driver is a program that is implemented independent from T-Kernel itself to control a hardware device or perform I/O processing with the hardware device. Since the difference of specifications among individual devices is absorbed by the device driver when an application or middleware operates a device or performs I/O processing with the device via the device driver, the application or middleware can enhance its hardware independency and compatibility.

Device management functions include a function to define a device driver, or to register the device driver to T-Kernel, and a function to use the registered device driver from an application or middleware.

While this registration of device drivers is mostly performed in the initialization at system startup, it can also be performed dynamically during the normal operation of the system. A device driver is registered in the device registration information (`ddev`) that is one of parameters for the extended SVC, tk_def_dev(), by specifying the set of functions (driver processing functions) of a program that actually implements device driver. These functions include the open function (openfn) that is called when a device is opened, the execute function (execfn) that is called when read or write processing starts, wait-for-completion function (waitfn) that waits for completion of read or write processing, etc. The actual operation of a device or I/O processing with the devices are performed in these driver processing functions.

As these driver processing functions are executed at protection level 0 as quasi-task portion, they can also access hardware directly. I/O processing with a device may be performed directly in these driver processing functions or may be performed in another task that runs based on the request from one of these driver processing functions. The specification of parameters, etc. when these driver processing functions are called is specified as a device driver interface. The device driver interface is an interface between a device driver and the T-Kernel device management functions.

When a device driver program is implemented, it is recommended to separate three layers of interface, logical, and physical layers carefully in order to enhance their maintainability and portability. The interface layer is responsible for implementing an interface between the T-Kernel device management functions and a device driver. The logical layer is responsible for performing a common processing according to the type of device. The physical layer is responsible for performing an operation dependent on the actual hardware or control chip. The interface specification, however, among the interface layer, logical layer, and physical layer is not specified in the T-Kernel, so that the actual layer separation can be implemented appropriately in each device driver. Programs that process the interface layer may be provided as libraries since there are many common processings that are independent of individual devices in the physical layer.

Extended SVCs are provided such as open (tk_opn_dev()), close (tk_cls_dev()), read (tk_rea_dev()), write (tk_wri_dev()), etc. to use the registered device driver from an application or middleware. The specification of these extended SVCs is called an application interface. For example, when an application executes tk_opn_dev() to open a device, the T-Kernel calls the open function (openfn) for the corresponding device driver to request the device open processing.

The positioning and configuration of T-Kernel device management functions are shown in Figure 5.1, “Device Management Functions”.

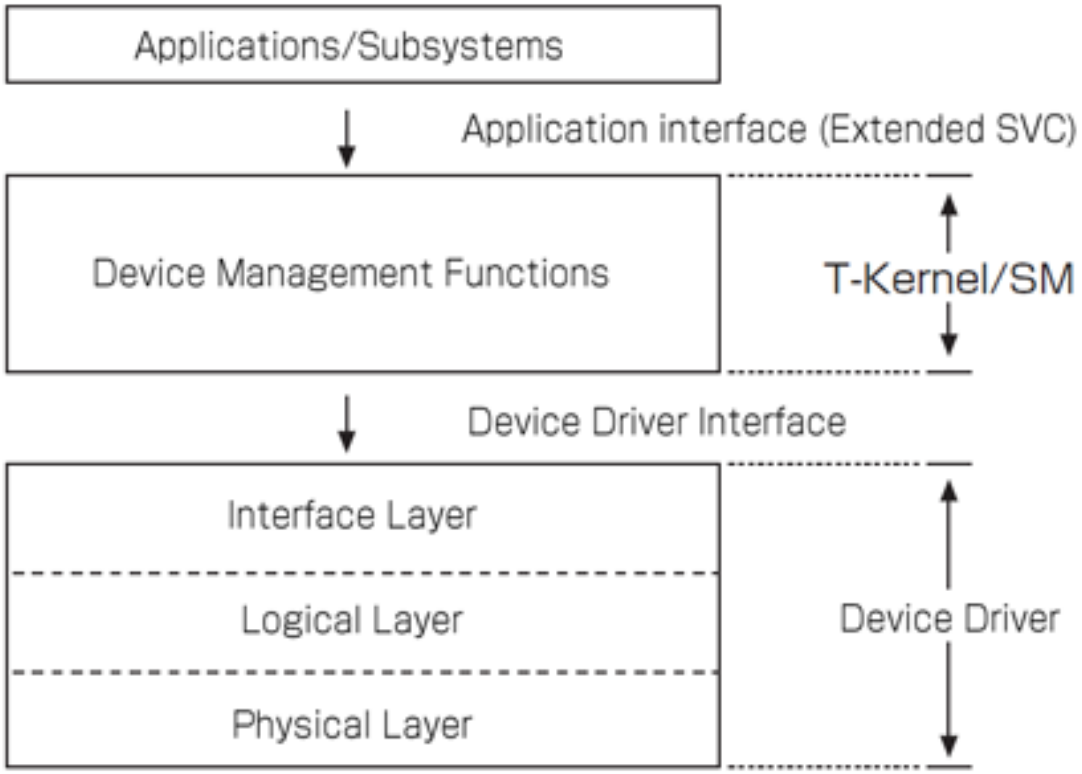

Figure 5.1: Device Management Functions

Additional Notes

The device drivers have common features with the subsystems as being implemented independent from T-Kernel itself and also being a system program to add or extend functions for T-Kernel. Additionally, both are also same in that they are both executed by loading the program into the system space, operate at protection level 0, and can access a hardware. While API for calling a device driver is limited to using open/close and read/write type, API for calling a subsystem can be defined without any restriction. The subsystems have functions to manage resources but the device drivers have no functions to do so.

Though T-Kernel device drivers managed by device management functions are assumed to be drivers for physical devices or hardware, they are not necessarily required to handle real physical devices or hardware. Also, system program for operating a device could be implemented as a subsystem rather than a device driver if it is not compatible with open/close or read/write type APIs.

5.3.1 Common Notes Related to Device Drivers

5.3.1.1 Basic Concepts

In addition to a physical device that represents a device as a physical hardware, there is a logical device that represents a perceived unit of a device from the viewpoint of software.

Although both devices match for most devices, when partitions were created on a hard disk or any other storage type device (SD card, USB storage, etc.), entire device represents a physical device and each partition represents a logical device.

The physical devices of same type are identified by "unit" while logical devices in one physical device are identified by "subunit." For example, the information that distinguishes the first hard disk from the second is called "unit," and the information that distinguishes the first partition from the second within that first hard disk is called "subunit."

The data definitions used in device management functions are explained in the subsequent subsections.

5.3.1.1.1 Device Name (UB* type)

A device name is a string of up to eight characters that is given to each device. It consists 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 of type + unit + subunit. Some devices may not have a unit or subunit, in which case the corresponding field is omitted.

The subunit is usually used to distinguish partitions in a hard disk. In other devices also, it can be used to create multiple logical devices in one physical device.

A name consisting of type + unit is called a physical device name. A name consisting of type + unit + subunit is called a logical 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.

Example 5.1 Example of Device Name

Device name	Target device
hda	Hard disk (entire disk)
hda0	Hard disk (1st partition)
fda	Floppy disk
rsa	Serial port
kbpd	Keyboard/pointing device

5.3.1.1.2 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)
```

Example 5.2 Example of Device ID

Device name	Device ID	Summary description
hda	devid	Hard disk (entire disk)
hda0	devid + 1	1st partition of hard disk
hda1	devid + 2	2nd partition of hard disk

5.3.1.1.3 Device Attribute (ATR type)

Device attributes are defined in order to represent a feature for each device and classify a device for each type. Device attributes should be specified when registering a device driver.

The specification method of device attributes is as follows:

```
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 protected */
#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 */
```

As to the above shown device type, whether it is the disk type (`TDK_DISK`) or not affects the processing procedure at the time of suspend. For more details, see the description of tk_sus_dev and Section 5.3.3.5, "Special Properties of Disk Devices".

Within the realm of T-Kernel, the device type other than disk type is not defined. Defining the device type other than disk type does not affect the behavior of T-Kernel. Other devices are assigned to undefined type (`TDK_UNDEF`).

For the disk device, the disk kinds are additionally defined. The typical disk kinds are as follows: For disk types other than these, see the specification related to device drivers or Section 7.1.1, "Disk Kind for Device Attributes" in Section 7.1, "Specification Related to Device Drivers to be Used as Reference".

```
 /* disk kind */
#define TDK_DISK_UNDEF  0x0010  /* miscellaneous disk */
#define TDK_DISK_HD     0x0015  /* hard disk */
#define TDK_DISK_CDROM  0x0016  /* CD-ROM */
```

The definition of disk kinds does not affect the T-Kernel behavior. These definitions are used only when they are required in a device driver or an application. For example, when an application must change its processing according to the kind of devices or media, the disk kind information is used. Devices or media that do not need such distinctions do not have to be assigned a device type.

5.3.1.1.4 Device Descriptor (ID type)

A device descriptor is an identifier used to access a device.

The device descriptor is assigned a positive value (> 0) by the T-Kernel/SM when a device is opened.

The device descriptor belongs to the same resource group as that of the task that opened the device. 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.

5.3.1.1.5 Request ID (ID type)

When an I/O request is made to a device, a request ID (> 0) is assigned identifying the request. This ID can be used to wait for I/O completion.

5.3.1.1.6 Data Number (W type, D type)

Data input/output from/to device is specified by a data number. Data is roughly classified into device-specific data and attribute data.

Device-specific data: Data number ≧ 0

As device-specific data, the data numbers are defined separately for each device.

Example 5.3 Example of Device-specific Data

device	Data number
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. For more details, see Section 5.3.1.2, “Attribute Data”.

5.3.1.2 Attribute Data

Attribute data are classified broadly into the following three types 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 individually for each device (device driver).

For the device kind attributes and device-specific attributes, see the specifications related to device driver. 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 the common attributes. If an unsupported data number is specified, error code E_PAR is returned.

The definition of common attributes is as follows:

```
#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 */
#define TDN_PCMCIAINFO  (-4)    /* R: PC card information */
#define TDN_DISKINFO_D  (-5)    /* R: disk information (64-bit device) */
```

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.

  As a device is registered by tk_def_dev when a device driver is started and the system default event notification message buffer ID (**evtmbfid**) is returned as this API return parameter, the value is held in the device driver and is used as the initial value of this attribute data.

  If 0 is set, device events are not notified. For device event notification, see Section 5.3.3.3, "Device Event Notification".

`TDN_DISKINFO`
: 32-bit device and disk information

  Data type DiskInfo

```
typedef enum {
        DiskFmt_STD     = 0,            /* standard (HD, etc.) */
        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       blockcont;              /* total block count */
} DiskInfo;
```

For definition of DiskFormat other than the above description, see the specification related to device drivers or Section 7.1.2, "Device Attribute Data" in Section 7.1, "Specification Related to Device Drivers to be Used as Reference".

`TDN_DISPSPEC`
Display Device Specification

Data type DEV_SPEC

For the definition of DEV_SPEC, see the specification related to device drivers or Section 7.1.2, "Device Attribute Data" in Section 7.1, "Specification Related to Device Drivers to be Used as Reference".

`TDN_DISKINFO_D`
64-bit device and disk information

Data type DiskInfo_D

```
typedef struct diskinfo_d {
        DiskFormat format;      /* format */
        BOOL    protect:1;      /* protected status */
        BOOL    removable:1;    /* removable */
        UW      rsv:30;         /* reserved (0) */
        W       blocksize;      /* block size in bytes */
        D       blockcont_d;    /* total number of blocks in 64-bit */
} DiskInfo_D;
```

Difference between DiskInfo_D and DiskInfo is only the part of their names being `blockcont` or `blockcont_d`, and the data type.

T-Kernel/SM does not convert a data between DiskInfo and DiskInfo_D. Both `TDN_DISKINFO` and `TDN_DISKINFO_D` just pass the request to device driver without any modification.

A disk driver must support `TDN_DISKINFO` and/or `TDN_DISKINFO_D`. It is recommended that `TDN_DISKINFO` is supported wherever possible for compatibility with T-Kernel 1.0.

Even if the total number of blocks of entire disk exceeds W, the number of blocks of individual partition may fit within W. In that case, the preferable implementation is such that a partitions fitting within W correspond to `TDN_DISKINFO` and partitions not fitting within W are determined to be an error (E_PAR) by `TDN_DISKINFO`. It is also preferable that `TDN_DISKINFO_D` is supported even if the number of blocks fit within W.

There is no direct dependency between the support for `TDN_DISKINFO_D` and the device driver attribute `TDA_DEV_D`. A device driver does not always have `TDA_DEV_D` attribute even if `TDN_DISKINFO_D` is supported. Also, `TDN_DISKINFO_D` is not always supported even if the device driver has `TDA_DEV_D` attribute.

As the definition of common attributes described above is a part of the specification of device driver rather than T-Kernel, it does not directly affect the T-Kernel behavior. Each device driver does not need to implement all the functions defined in the common attributes. However, as the definition of common attributes is applicable to all the device drivers, the specification of each device driver must be specified in a way that does not conflict with this definitions.

Difference from T-Kernel 1.0
Attribute data for `TDN_DISKINFO_D` is added to support 64-bit devices.

5.3.2 Device Input/Output Operations

The application interface is used to make use of the registered device drivers from an application or middleware. 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( CONST UB *devnm, UINT omode )
ER  tk_cls_dev( ID dd, UINT option )
ID  tk_rea_dev( ID dd, W start, void *buf, W size, TMO tmout )
ID  tk_rea_dev_du( ID dd, D start_d, void *buf, W size, TMO_U tmout_u )
ER  tk_srea_dev( ID dd, W start, void *buf, W size, W *asize )
ER  tk_srea_dev_d( ID dd, D start_d, void *buf, W size, W *asize )
ID  tk_wri_dev( ID dd, W start, CONST void *buf, W size, TMO tmout )
ID  tk_wri_dev_du( ID dd, D start_d, CONST void *buf, W size, TMO_U tmout_u )
ER  tk_swri_dev( ID dd, W start, CONST void *buf, W size, W *asize )
ER  tk_swri_dev_d( ID dd, D start_d, CONST void *buf, W size, W *asize )
ID  tk_wai_dev( ID dd, ID reqid, W *asize, ER *ioer, TMO tmout )
ID  tk_wai_dev_u( ID dd, ID reqid, W *asize, ER *ioer, TMO_U tmout_u )
INT tk_sus_dev( UINT mode )
ID  tk_get_dev( ID devid, UB *devnm )
ID  tk_ref_dev( CONST 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, void *evtinf )
```

5.3.2.1 tk_opn_dev - Open Device

C Language Interface

```
#include <tk/tkernel.h>
```

ID dd = tk_opn_dev (CONST UB *devnm , UINT omode);

Parameter

CONST UB*	devnm	Device Name	Device name
UINT	omode	Open Mode	Open mode

Return Parameter

ID	dd	Device Descriptor	Device descriptor
	or	Error Code	Error code

Error Code

E_BUSY	Device BUSY (exclusive open)
E_NOEXS	Device does not exist
E_LIMIT	Open count exceeds the limit
Other	Error code returned by device driver

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

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          /* unnecessary to be locked (resident) */
```

TD_READ
: read only

TD_WRITE
: Write only

`TD_UPDATE`
: Read/write

  Sets the 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.

Present Open Mode		Concurrent Open Mode											
		No exclusive mode			TD_WEXCL			TD_REXCL			TD_EXCL		
		R	U	W	R	U	W	R	U	W	R	U	W
No exclusive mode	R	YES	YES	YES	YES	YES	YES	NO	NO	NO	NO	NO	NO
	U	YES	YES	YES	NO	NO	NO	NO	NO	NO	NO	NO	NO
	W	YES	YES	YES	NO	NO	NO	YES	YES	YES	NO	NO	NO
TD_WEXCL	R	YES	NO	NO	YES	NO	NO	NO	NO	NO	NO	NO	NO
	U	YES	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO
	W	YES	NO	NO	NO	NO	NO	YES	NO	NO	NO	NO	NO
TD_REXCL	R	NO	NO	YES	NO	NO	YES	NO	NO	NO	NO	NO	NO
	U	NO	NO	YES	NO	NO	NO	NO	NO	NO	NO	NO	NO
	W	NO	NO	YES	NO	NO	NO	NO	NO	YES	NO	NO	NO
TD_EXCL	R	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO
	U	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO
	W	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO

Table 5.1: Whether Concurrent Open of Same Device is Allowed or NOT

```
R = TD_READ
W = TD_WRITE
U = TD_UPDATE
YES = Yes, can be opened
NO = No, cannot be opened (E_BUSY)
```

`TD_NOLOCK`
: unnecessary to be locked (resident)

Indicates that a memory space (`buf`) specified in I/O 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.

5.3.2.2 tk_cls_dev - Close Device

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_cls_dev (ID dd , UINT option);

Parameter

ID	dd	Device Descriptor	Device descriptor
UINT	option	Close Option	Close option

Return Parameter

ER	ercd	Error Code	Error code

Error Code

E_ID	dd is invalid or not open
Other	Error code returned by device driver

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

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.

5.3.2.3 tk_rea_dev - Start Read Device

C Language Interface

```
#include <tk/tkernel.h>
```

ID reqid = tk_rea_dev (ID dd , W start , void *buf , W size , TMO tmout);

Parameter

ID	dd	Device Descriptor	Device descriptor
W	start	Start Location	Read start location (≧ 0: Device-specific data, < 0: Attribute data)
void*	buf	Buffer	Buffer location for putting the read data
W	size	Read Size	Read size
TMO	tmout	Timeout	Request acceptance timeout (ms)

Return Parameter

ID	reqid		Request ID	Request ID
		or	Error Code	Error code

Error Code

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	Error code returned by device driver

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Initiates reading device-specific data or attribute data from the specified device. This function initiates 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 initiating read operation differs among device drivers; return of control is not necessarily immediate.

In the case of device-specific data, the start and size units are defined 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 in tmout. Note that the timeout applies to the request acceptance. Once a request has been accepted, this function does not time out.

This extended SVC can be used for a device driver that has the TDA_DEV_D or TDA_TMO_U attribute. In that case, the parameters are converted appropriately by T-Kernel/SM. For example, when a device driver has the TDA_TMO_U attribute, the timeout interval (milliseconds) specified in tmout of this extended SVC is converted to the time in microseconds, and then passed to the device driver with the TDA_TMO_U attribute.

Difference from T-Kernel 1.0

The data type of start and size was changed from INT to W. This is because it is more easier to understand to fix the number of bits at a known value for the parameters closely related to the functions (time management and device management) that now have the 64-bit specifications in T-Kernel 2.0. The reason why the type of MSEC and TMO was changed from INT to W, and the type of RELTIM was changed from UINT to UW is also similar, in addition to the relationship with μT-Kernel.

5.3.2.4 tk_rea_dev_du - Read Device (in 64-bit microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

ID reqid = tk_rea_dev_du (ID dd , D start_d , void *buf , W size , TMO_U tmout_u);

Parameter

ID	dd	Device Descriptor	Device descriptor
D	start_d	Start Location	Read start location (64 bit, ≧ 0: Device-specific data, < 0: Attribute data)
void*	buf	Buffer	Buffer location for putting the read data
W	size	Read Size	Read size
TMO_U	tmout_u	Timeout	Request acceptance timeout (in microseconds)

Return Parameter

ID	reqid		Request ID	Request ID
		or	Error Code	Error code

Error Code

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	Error code returned by device driver

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

This extended SVC takes the parameters start_d (64 bits) and tmout_u (64-bit microseconds), instead of the parameters start and tmout of tk_rea_dev.

Its specification is the same as that of tk_rea_dev, except that the parameters are changed to start_d and tmout_u. For more details, see the description of tk_rea_dev.

Additional Notes

If the corresponding device driver does not have the TDA_DEV_D attribute, the error code E_PAR is returned when specifying a value that is out of the range of W for the start position start_d.

If the corresponding device driver does not have the TDA_TMO_U attribute (does not supports microseconds), it cannot handle the timeout in microseconds. In that case, the timeout (in microseconds) specified by this extended SVC in tmout_u is rounded to the time in milliseconds and passed to the device driver.

Thus, the appropriate conversion of parameters is executed by T-Kernel/SM. The application does not have to know whether the device driver has the TDA_DEV_D attribute or not, or whether the device driver supports 64 bits or not.

Difference from T-Kernel 1.0

This extended SVC was added in T-Kernel 2.0.

tk_rea_dev_du and tk_wri_dev_du include the both meanings of the suffixes, '_u' and '_d', because their start positions are 64 bits and timeouts are 64-bit microseconds.

5.3.2.5 tk_srea_dev - Synchronous Read

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_srea_dev (ID dd , W start , void *buf , W size , W *asize);

Parameter

ID	dd	Device Descriptor	Device descriptor
W	start	Start Location	Read start location (≧ 0: Device-specific data, < 0: Attribute data)
void*	buf	Buffer	Buffer location for putting the read data
W	size	Read Size	Read size
W*	asize	Actual Size	Pointer to the area to return the read size

Return Parameter

ER	ercd	Error Code	Error code
W	asize	Actual Size	Actually read size

Error Code

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_ABORT	Processing aborted
Other	Error code returned by device driver

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Synchronous read. This is equivalent to the following.

```
ER tk_srea_dev( ID dd, W start, void *buf, W size, W *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 = ioer;
        }

        return er;
}
```

This extended SVC can be used for a device driver that has the TDA_DEV_D attribute. In that case, the parameters are converted appropriately by T-Kernel/SM.

Difference from T-Kernel 1.0

The data type of start and size is changed from INT to W, and the data type of asize is changed from INT* to W*.

5.3.2.6 tk_srea_dev_d - Synchronous Read (64 bit)

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_srea_dev_d (ID dd , D start_d , void *buf , W size , W *asize);

Parameter

ID	dd	Device Descriptor	Device descriptor
D	start_d	Start Location	Read start location (64 bit, ≧ 0: Device-specific data, < 0: Attribute data)
void*	buf	Buffer	Buffer location for putting the read data
W	size	Read Size	Read size
W*	asize	Actual Size	Pointer to the area to return the read size

Return Parameter

ER	ercd	Error Code	Error code
W	asize	Actual Size	Actually read size

Error Code

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_ABORT	Processing aborted
Other	Error code returned by device driver

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

This extended SVC takes the 64-bit parameter start_d, instead of the parameter start of tk_srea_dev.

Its specification is the same as that of tk_srea_dev, except that the parameter is changed to start_d. For more details, see the description of tk_srea_dev.

Additional Notes

If the corresponding device driver does not have the TDA_DEV_D attribute, the error code E_PAR is returned when specifying a value that is out of the range of W for the start position start_d.

Thus, the appropriate conversion of parameters is executed by T-Kernel/SM. The application does not have to know whether the device driver has the TDA_DEV_D attribute, or whether the device driver supports 64 bits.

Difference from T-Kernel 1.0

This extended SVC was added in T-Kernel 2.0.

5.3.2.7 tk_wri_dev - Start Write Device

C Language Interface

```
#include <tk/tkernel.h>
```

ID reqid = tk_wri_dev (ID dd , W start , CONST void *buf , W size , TMO tmout);

Parameter

ID	dd	Device Descriptor	Device descriptor
W	start	Start Location	write start location (≧ 0: Device-specific data, < 0: Attribute data)
CONST void*	buf	Buffer	Buffer holding data to be written
W	size	Write Size	Size of data to be written
TMO	tmout	Timeout	Request acceptance timeout (ms)

Return Parameter

ID	reqid	Request ID	Request ID
	or	Error Code	Error code

Error Code

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	Error code returned by device driver

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Initiates writing device-specific data or attribute data to a device. This function initiates 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 initiating write operation differs among device drivers; return of control is not necessarily immediate.

In the case of device-specific data, the start and size units are defined 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 in tmout. Note that the timeout applies to the request acceptance. Once a request has been accepted, this function does not time out.

This extended SVC can be used for a device driver that has the TDA_DEV_D or TDA_TMO_U attribute. In that case, the parameters are converted appropriately by T-Kernel/SM. For example, when a device driver has the TDA_TMO_U attribute, the timeout interval (milliseconds) specified in tmout of this extended SVC is converted to the time in microseconds, and then passed to the device driver with the TDA_TMO_U attribute.

Difference from T-Kernel 1.0

The data type of start and size was changed from INT to W.

5.3.2.8 tk_wri_dev_du - Write Device (in 64-bit microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

ID reqid = tk_wri_dev_du (ID dd , D start_d , CONST void *buf , W size , TMO_U tmout_u);

Parameter

ID	dd	Device Descriptor	Device descriptor
D	start_d	Start Location	Write start location (64 bit, ≧ 0: Device-specific data, < 0: Attribute data)
CONST void*	buf	Buffer	Buffer holding data to be written
W	size	Write Size	Size of data to be written
TMO_U	tmout_u	Timeout	Request acceptance timeout (in microseconds)

Return Parameter

ID	reqid	Request ID	Request ID
	or	Error Code	Error code

Error Code

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	Error code returned by device driver

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

This extended SVC takes the parameters start_d (64 bits) and tmout_u (64-bit microseconds), instead of the parameters start and tmout of tk_wri_dev.

Its specification is the same as that of tk_wri_dev, except that the parameters are changed to start_d and tmout_u. For more details, see the description of tk_wri_dev.

Additional Notes

If the corresponding device driver does not have the TDA_DEV_D attribute, the error code E_PAR is returned when specifying a value that is out of the range of W for the start position start_d.

If the corresponding device driver does not have the TDA_TMO_U attribute (does not supports microseconds), it cannot handle the timeout in microseconds. In that case, the timeout (in microseconds) specified by this extended SVC in tmout_u is rounded to the time in milliseconds and passed to the device driver.

Thus, the appropriate conversion of parameters is executed by T-Kernel/SM. The application does not have to know whether the device driver has the TDA_DEV_D attribute or not, or whether the device driver supports 64 bits or not.

Difference from T-Kernel 1.0

This extended SVC was added in T-Kernel 2.0.

tk_rea_dev_du and tk_wri_dev_du include the both meanings of the suffixes '_u' and '_d', because their start positions are 64 bits and timeouts are 64-bit microseconds.

5.3.2.9 tk_swri_dev - Synchronous Write

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_swri_dev (ID dd , W start , CONST void *buf , W size , W *asize);

Parameter

ID	dd	Device Descriptor	Device descriptor
W	start	Start Location	Write start location (≧ 0: Device-specific data, < 0: Attribute data)
CONST void*	buf	Buffer	Buffer holding data to be written
W	size	Write Size	Size of data to be written
W*	asize	Actual Size	Pointer to the area to return the written size

Return Parameter

ER	ercd	Error Code	Error code
W	asize	Actual Size	Actually written size

Error Code

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_ABORT	Processing aborted
Other	Error code returned by device driver

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Synchronous write. This is equivalent to the following.

```
ER tk_swri_dev( ID dd, W start, void *buf, W size, W *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;
```

```
}
```

This extended SVC can be used for a device driver that has the `TDA_DEV_D` attribute. In that case, the parameters are converted appropriately by T-Kernel/SM.

Difference from T-Kernel 1.0

The data type of `start` and `size` is changed from INT to W, and the data type of `asize` is changed from INT* to W*.

5.3.2.10 tk_swri_dev_d - Synchronous Write (64 bit)

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_swri_dev_d (ID dd , D start_d , CONST void *buf , W size , W *asize);

Parameter

ID	dd	Device Descriptor	Device descriptor
D	start_d	Start Location	Write start location (64 bit, ≧ 0: Device-specific data, < 0: Attribute data)
CONST void*	buf	Buffer	Buffer holding data to be written
W	size	Write Size	Size of data to be written
W*	asize	Actual Size	Pointer to the area to return the written size

Return Parameter

ER	ercd	Error Code	Error code
W	asize	Actual Size	Actually written size

Error Code

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_ABORT	Processing aborted
Other	Error code returned by device driver

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

This extended SVC takes the 64-bit parameter start_d, instead of the parameter start of tk_swri_dev.

Its specification is the same as that of tk_swri_dev, except that the parameter is changed to start_d. For more details, see the description of tk_swri_dev.

Additional Notes

If the corresponding device driver does not have the TDA_DEV_D attribute, the error code E_PAR is returned when specifying a value that is out of the range of W for the start position start_d.

Thus, the appropriate conversion of parameters is executed by T-Kernel/SM. The application does not have to know whether the device driver has the TDA_DEV_D attribute or not, or whether the device driver supports 64 bits or not.

Difference from T-Kernel 1.0

This extended SVC was added in T-Kernel 2.0.

5.3.2.11 tk_wai_dev - Wait for Request Completion for Device

C Language Interface

```
#include <tk/tkernel.h>
```

ID creqid = tk_wai_dev (ID dd , ID reqid , W *asize , ER *ioer , TMO tmout);

Parameter

ID	dd	Device Descriptor	Device descriptor
ID	reqid	Request ID	Request ID
W*	asize	Actually Read/Written Size	Pointer to the area to return the read/written size
ER*	ioer	I/O Error	Pointer to the area to return I/O error
TMO	tmout	Timeout	Timeout (ms)

Return Parameter

ID	creqid		Completed Request ID	Completed request ID
		or	Error Code	Error code
W	asize		Actually Read/Written Size	Actually read/written size
ER	ioer		I/O Error	I/O error

Error Code

E_ID	dd is invalid or not opened, or 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	Error code returned by device driver

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

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 still ongoing. When `reqid` > 0 and `tmout` = `TMO_FEVR` are both set, the processing must be completed without timing out.

If the device driver returns a processing result error (such as I/O error) for the requested processing, the error code is stored in `ioer` instead of the return code. Specifically, the error code, which is stored in `error` of the request packet T_DEVREQ by the wait-for-completion function (waitfn) called for processing tk_wai_dev, is returned to `ioer` as the processing result error.

On the other hand, the return code is used for errors when the wait request itself was not handled properly. When error is passed in the return code, `ioer` has no meaning. Note also that if an error is passed in the return code, tk_wai_dev() must be called again to wait for completion since the processing is still ongoing. For more details, see Section 5.3.3.2.4, "waitfn - Wait-for-completion function".

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 `dd`. Similarly, if there is a task waiting for request completion with `reqid` > 0 set, another task cannot wait for completion specifying `reqid` = 0.

This extended SVC can be used for a device driver that has the `TDA_TMO_U` attribute. In that case, the parameters are converted appropriately by T-Kernel/SM. For example, when a device driver has the `TDA_TMO_U` attribute, the timeout interval (milliseconds) specified in `tmout` of this extended SVC is converted to the time in microseconds, and then passed to the device driver with the `TDA_TMO_U` attribute.

Difference from T-Kernel 1.0

The data type of `asize` was changed from INT* to W*.

5.3.2.12 tk_wai_dev_u - Wait Device (in microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

ID creqid = tk_wai_dev_u (ID dd , ID reqid , W *asize , ER *ioer , TMO_U tmout_u);

Parameter

ID	dd	Device Descriptor	Device descriptor
ID	reqid	Request ID	Request ID
W*	asize	Actually Read/Written Size	Pointer to the area to return the read/written size
ER*	ioer	I/O Error	Pointer to the area to return I/O error
TMO_U	tmout_u	Timeout	Timeout (in microseconds)

Return Parameter

ID	creqid		Completed Request ID	Completed request ID
		or	Error Code	Error code
W	asize		Actually Read/Written Size	Actually read/written size
ER	ioer		I/O Error	I/O error

Error Code

E_ID	dd is invalid or not opened, or 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	Error code returned by device driver

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

This extended SVC takes the parameter tmout_u (64-bit microseconds), instead of the parameter tmout of tk_wai_dev.

Its specification is the same as that of tk_wai_dev, except that the parameter changed to tmout_u. For more details, see the description of tk_wai_dev.

Additional Notes

If the corresponding device driver does not have the TDA_TMO_U attribute (does not supports microseconds), it cannot handle the timeout in microseconds. In that case, the timeout (in microseconds) specified by this extended SVC in tmout_u is rounded to the time in milliseconds and passed to the device driver.

Thus, the appropriate conversion of parameters is executed by T-Kernel/SM. The application does not have to know whether the device driver has the `TDA_TMO_U` attribute or not, or whether the device driver supports microseconds or not.

Difference from T-Kernel 1.0

This extended SVC was added in T-Kernel 2.0.

Note that an extended SVC of device management function tk_wai_dev_u is appended with the suffix '_u', not '_d'.

5.3.2.13 tk_sus_dev - Suspends Device

C Language Interface

```
#include <tk/tkernel.h>

INT dissus = tk_sus_dev (UINT mode );
```

Parameter

UINT	mode	Mode	Mode

Return Parameter

INT	dissus	Suspend Disable Request Count	Suspend disable request count
	or	Error Code	Error code

Error Code

E_BUSY	Suspend already disabled
E_QOVR	Suspend disable request count limit exceeded

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

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      /* forced suspend specification */
```

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. Suspended state
   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)

In the above processing, whether the device is a disk device or not is determined by checking whether the device attribute is the disk type (TDK_DISK) or not.

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.

5.3.2.14 tk_get_dev - Get Device Name

C Language Interface

```
#include <tk/tkernel.h>
```

ID pdevid = tk_get_dev (ID devid , UB *devnm);

Parameter

ID	devid	Device ID	Device ID
UB*	devnm	Device Name	Pointer to the device name storage location

Return Parameter

ID	pdevid		Device ID of Physical Device	Device ID of the physical device
		or	Error Code	Error code
UB	devnm		Device Name	Device name

Error Code

E_NOEXS	The device specified in devid does not exist

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

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.

5.3.2.15 tk_ref_dev - Get Device Information

C Language Interface

```
#include <tk/tkernel.h>
```

ID devid = tk_ref_dev (CONST UB *devnm , T_RDEV *rdev);

Parameter

CONST UB*	devnm	Device Name	Device name
T_RDEV*	rdev	Packet to Return Device Information	Pointer to the area to return the device information

Return Parameter

ID	devid	Device ID	Device ID
	or	Error Code	Error code

rdev Detail:

ATR	devatr	Device Attribute	Device attributes
INT	blksz	Block Size of Device-specific Data	Block size of device-specific data (-1: unknown)
INT	nsub	Subunit Count	Number of subunits
INT	subno	Subunit Number	0: Physical device, 1 to nsub: Subunit number+1

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

Error Code

E_NOEXS	The device specified in devnm does not exist

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Gets device information about the device specified in devnm, 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.

The device ID of the device specified in devnm is passed in the return code.

5.3.2.16 tk_oref_dev - Get Device Information

C Language Interface

```
#include <tk/tkernel.h>
```

ID devid = tk_oref_dev (ID dd , T_RDEV *rdev);

Parameter

ID	dd	Device Descriptor	Device descriptor
T_RDEV*	rdev	Packet to Return Device Information	Pointer to the area to return the device information

Return Parameter

ID	devid	Device ID	Device ID
	or	Error Code	Error code

rdev Detail:

ATR	devatr	Device Attribute	Device attributes
INT	blksz	Block Size of Device-specific Data	Block size of device-specific data (-1: unknown)
INT	nsub	Subunit Count	Number of subunits
INT	subno	Subunit Number	0: Physical device, 1 to nsub: Subunit number+1

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

Error Code

E_ID	dd is invalid or not open

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Gets device information about the device specified in 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 dd.

The device ID of the device specified in dd is passed in the return code.

5.3.2.17 tk_lst_dev - Get Registered Device Information

C Language Interface

```
#include <tk/tkernel.h>

INT remcnt = tk_lst_dev (T_LDEV *ldev , INT start , INT ndev );
```

Parameter

T_LDEV*	ldev	List of Devices	Location of registered device information (array)
INT	start	Starting Number	Starting number
INT	ndev	Number of Devices	Number to acquire

Return Parameter

INT	remcnt	Remaining Device Count	Number of remaining registrations
	or	Error Code	Error code

ldev Detail:

ATR	devatr	Device Attribute	Device attributes
INT	blksz	Block Size of Device-specific Data	Block size of device-specific data (-1: unknown)
INT	nsub	Subunit Count	Number of subunits
UB	devnm[L_DEVNM]	Physical Device Name	Physical device name

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

Error Code

E_NOEXS	start exceeds the registered number

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

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 be always obtained if the acquisition is carried out over multiple operations.

5.3.2.18 tk_evt_dev - Send Driver Request Event to Device

C Language Interface

```
#include <tk/tkernel.h>
```

INT retcode = tk_evt_dev (ID devid , INT evttyp , void *evtinf);

Parameter

ID	devid	Device ID	Event destination device ID
INT	evttyp	Event Type	Driver request event type
void*	evtinf	Event Information	Information for each event type

Return Parameter

INT	retcode		Return Code from eventfn	Return code passed by eventfn
		or	Error Code	Error code

Error Code

E_NOEXS	The device specified in devid does not exist
E_PAR	Internal device manager events (evttyp < 0) cannot be specified
Other	Error code returned by device driver

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Sends a driver request event to the device (device driver) specified in devid.

The functioning of driver request events and the contents of evtinf are defined for each event type. For details on driver request event, see Section 5.3.3.2.6, “eventfn - Event function”.

5.3.3 Registration of Device Driver

5.3.3.1 Registration Method of Device Driver

Device driver registration is performed for each physical device.

5.3.3.1.1 tk_def_dev - Register Device

C Language Interface

```
#include <tk/tkernel.h>
```

ID devid = tk_def_dev (CONST UB *devnm , CONST T_DDEV *ddev , T_IDEV *idev);

Parameter

CONST UB*	devnm	Physical Device Name	Physical device name
CONST T_DDEV*	ddev	Define Device	Device registration information
T_IDEV*	idev	Initial Device Information	Device initial information

Return Parameter

ID	devid		Device ID	Device ID
		or	Error Code	Error code

idev Detail:

ID	evtmbfid	Event Notification Message Buffer ID	Event notification message buffer ID

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

Error Code

E_LIMIT	Number of registrations exceeds the system limit
E_NOEXS	The device specified in devnm does not exist (when ddev = NULL)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Registers a device (device driver) with the device name set in devnm, and passes the device ID of the registered device in the return code. 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.

ddev specifies the device registration information. When ddev = NULL is specified, device devnm registration is deleted.

ddev is a structure in the following format:

```
typedef struct t_ddev {
        void    *exinf;   /* extended information */
        ATR     drvatr;   /* driver attributes */
        ATR     devatr;   /* device attributes */
        INT     nsub;     /* number of subunits */
        INT     blksz;    /* block size of device-specific data (-1: unknown) */
        FP      openfn;   /* open function */
```

```
        FP      closefn;  /* close function */
        FP      execfn;   /* execute function */
        FP      waitfn;   /* wait-for-completion function */
        FP      abortfn;  /* abort function */
        FP      eventfn;  /* event function */
        /* Implementation-dependent information may be added beyond this point.*/
} T_DDEV;
```

exinf is used to store any desired information. The value is passed to the processing functions. Device management pays no attention to the contents.

drvatr sets device driver attribute information. The lower bits indicate system attributes, and the high bits are used for implementation-dependent attributes. The implementation-dependent attribute portion is used, for example, to define validity flags when implementation-dependent data is added to T_DDEV.

```
drvatr := [TDA_OPENREQ] | [TDA_TMO_U] | [TDA_DEV_D]
```

```
#define TDA_OPENREQ     0x0001  /* open/close each time */
#define TDA_TMO_U       0x0002  /* timeout in microseconds is used */
#define TDA_DEV_D       0x0004  /* 64 bit device */
```

drvatr can be specified by combining the following driver attributes.

TDA_OPENREQ

When a device is opened multiple times, normally openfn is called only 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.

TDA_TMO_U

Indicates that timeout in microseconds is used.

In this case, the timeout tmout of driver processing functions is specified in the TMO_U format (microseconds).

TDA_DEV_D

Indicates that a 64-bit device is used. In this case, the type of the request packet devreq of driver processing functions is T_DEVREQ_D.

If TDA_TMO_U or TDA_DEV_D is specified, type of some parameters of driver processing functions is changed. If a combination of multiple driver attributes that change the type of parameters is specified in a driver processing function, the type of all specified parameters of that function is changed.

Device attributes are specified in devatr. The details of device attribute setting 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 driver processing functions. For more details on driver processing functions, see Section 5.3.3.2, “Device Driver Interface”.

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.

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 registered or deleted physical device.

Device registration or update:
tk_evt_ssy(0, TSEVT_DEVICE_REGIST, 0, devid)

Device deletion:
tk_evt_ssy(0, TSEVT_DEVICE_DELETE, 0, devid)

Difference from T-Kernel 1.0

TDA_TMO_U and TDA_DEV_D are added as attributes of drvatr to support 64-bit devices.

5.3.3.1.2 tk_ref_idv - Reference Device Initialization Information

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_idv (T_IDEV *idev);

Parameter

T_IDEV*	idev	Packet to Return Initial Device Information	Pointer to the area to return the device initialization information

Return Parameter

ER	ercd	Error Code	Error code

idev Detail:

ID	evtmbfid	Event Notification Message Buffer ID	Event notification message buffer ID

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

Error Code

E_MACV	Memory access privilege error

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Gets device initialization information. The contents are the same as the information obtained by tk_def_dev().

Additional Notes

The error code E_MACV is common to many system calls, and usually not included in the error code list of each system call. However, for this extended SVC, E_MACV is included in this error code list because it is the only typical error.

5.3.3.2 Device Driver Interface

The device driver interface consists of processing functions (driver processing functions) specified when registering a device.

Open function
ER openfn(ID devid, UINT omode, void *exinf);

Close function
ER closefn(ID devid, UINT option, void *exinf);

Execute function
ER execfn(T_DEVREQ *devreq, TMO tmout, void *exinf);

Wait-for-completion function
INT waitfn(T_DEVREQ *devreq, INT nreq, TMO tmout, void *exinf);

Abort function
ER abortfn(ID tskid, T_DEVREQ *devreq, INT nreq, void *exinf);

Event function
INT eventfn(INT evttyp, void *evtinf, void *exinf);

If TDA_TMO_U is specified for a driver attribute, the timeout specification tmout for the following driver processing functions is set to TMO_U type (in microseconds).

Execute function
ER execfn(T_DEVREQ *devreq, TMO_U tmout_u, void *exinf);

Wait-for-completion function
INT waitfn(T_DEVREQ *devreq, INT nreq, TMO_U tmout_u, void *exinf);

If TDA_DEV_D is specified for a driver attribute, the type of request packet devreq for the following driver processing functions is set to T_DEVREQ_D.

Execute function
ER execfn(T_DEVREQ_D *devreq_d, TMO tmout, void *exinf);

Wait-for-completion function
INT waitfn(T_DEVREQ_D *devreq_d, INT nreq, TMO tmout, void *exinf);

Abort function
ER abortfn(ID tskid, T_DEVREQ_D *devreq_d, INT nreq, void *exinf);

If TDA_TMO_U and TDA_DEV_D are specified set a driver attribute, a driver processing function is set to the one that has parameters with all the specified types of changes were applied.

Execute function
ER execfn(T_DEVREQ_D *devreq_d, TMO_U tmout_u, void *exinf);

Wait-for-completion function
INT waitfn(T_DEVREQ_D *devreq_d, INT nreq, TMO_U tmout_u, void *exinf);

Driver processing functions are called by device management and run as a quasi-task portion. These driver processing functions must be reentrant. Calling of these driver processing functions in a mutually exclusive manner is not guaranteed. If, for example, there are simultaneous requests from multiple devices for the same device, different tasks might call the same driver processing function at the same time. The device driver must perform mutual exclusion control in such cases as necessary.

I/O requests to a device driver are made by means of the following request packet associated with a request ID.

```
/*
 * Device request packet: For 32-bit
 * In:  Input parameter to driver processing function (set in T-Kernel/SM device management )
 * Out: Output parameter from driver processing function (set in driver processing function )
 */
typedef struct t_devreq {
        struct t_devreq *next;    /* In: Link to request packet (NULL: termination) */
        void    *exinf;           /* X: Extended information */
        ID      devid;            /* In: Target device ID */
        INT     cmd:4;            /* In: Request command */
        BOOL    abort:1;          /* In: TRUE if abort request */
        BOOL    nolock:1;         /* In: TRUE if lock (making resident) not needed */
        INT     rsv:26;           /* In: Reserved (always 0) */
        T_TSKSPC tskspc;          /* In: Task space of requesting task */
        W       start;            /* In: Starting data number */
        W       size;             /* In: Request size */
        void    *buf;             /* In: IO buffer address */
        W       asize;            /* Out: Size of result */
        ER      error;            /* Out: Error result */
        /* Implementation-dependent information may be added beyond this point.*/
} T_DEVREQ;
```

```
/*
 * Device request packet: For 64-bit
 * In:  Input parameter to driver processing function (set in T-Kernel/SM device management )
 * Out: Output parameter from driver processing function (set in driver processing function )
 */
typedef struct t_devreq_d {
        struct t_devreq_d *next;  /* In: Link to request packet (NULL: termination) */
        void    *exinf;           /* X: Extended information */
        ID      devid;            /* In: Target device ID */
        INT     cmd:4;            /* In: Request command */
        BOOL    abort:1;          /* In: TRUE if abort request */
        BOOL    nolock:1;         /* In: TRUE if lock (making resident) not needed */
        INT     rsv:26;           /* In: Reserved (always 0) */
        T_TSKSPC tskspc;          /* In: Task space of requesting task */
        D       start_d;          /* In: Starting data number, 64-bit */
        W       size;             /* In: Request size */
        void    *buf;             /* In: IO buffer address */
        W       asize;            /* Out: Size of result */
        ER      error;            /* Out: Error result */
        /* Implementation-dependent information may be added beyond this point.*/
} T_DEVREQ_D;
```

In: Input parameter to the driver processing function is set in T-Kernel/SM device management. Should not be changed on the device driver side. Parameters other than input parameters (In) are initially cleared to 0 by the device management. After that, device management does not modify them.Out: Output parameter returned from the driver processing function is set in the driver processing function.

`next` is used to link the request packet. In addition to usage 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. Device management does not pay attention to the 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 incorrect operation if the device driver performs a lock. Accordingly, when nolock = TRUE, the device driver must not lock the space.)

tskspc is set as the task space for a task (API issuing task) that issued API for device I/O operation. Since the processing function is executed in a context of a quasi-task portion in which the API issuing task is a requesting task, tskspc is same as the task space for the processing function. If, however, the actual I/O 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 actual I/O processing to the task space of the task issuing API.

start, start_d, and size are just set as start, start_d, and size specified in tk_rea_dev(),

tk_rea_dev_du(), tk_wri_dev(), and tk_wri_dev_du().

buf is just set as buf specified in tk_rea_dev(), tk_rea_dev_du(), tk_wri_dev(), and tk_wri_dev_du(). 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.

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... are described above).

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.

Difference between T_DEVREQ and T_DEVREQ_D is only the part of their names being start or start_d, and the data type.

The type of device request packet (T_DEVREQ or T_DEVREQ_D) is selected based on the driver attribute (TDA_DEV_D) at device registration. For this reason, T_DEVREQ and T_DEVRE do not co-exist in the request packet for one driver.

Difference from T-Kernel 1.0
The data type of start, size, and asize for T_DEVREQ was changed from INT to W. Device request packet for T_DEVREQ_D is added to support 64-bit devices.

5.3.3.2.1 openfn - Open function

C Language Interface

ER ercd = openfn (ID devid , UINT omode , void *exinf);

Parameter

ID	devid	Device ID	Device ID of the device to open
UINT	omode	Open Mode	Open mode (same as tk_opn_dev)
void*	exinf	Extended Information	Extended information set at device registration

Return Parameter

ER	ercd	Error Code	Error code

Error Code

Other	Error code returned by the device driver

Description

The open function openfn is called when tk_opn_dev() is invoked.

The function openfn performs processing to enable 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 simply 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 can be performed so long as there is no problem in device driver operation.

When openfn is used to perform device initialization or the like, in principle no processing should be performed that causes a wait. 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. There is no need for openfn to initialize the device.

When the same device is opened multiple times, normally this function is called only for 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.

The openfn function does not need to perform any processing with regard to multiple opening or open mode, which are handled by device management. Likewise, omode is simply passed as reference information; no processing relating to omode is required.

openfn runs as a quasi-task portion of the task that issued tk_opn_dev. That is, it is executed in the context of the quasi-task portion whose requesting task is the task that issued tk_opn_dev.

5.3.3.2.2 closefn - Close function

C Language Interface

ER ercd = closefn (ID devid , UINT option , void *exinf);

Parameter

ID	devid	Device ID	Device ID of the device to close
UINT	option	Close Option	Close option (same as tk_cls_dev)
void*	exinf	Extended Information	Extended information set at device registration

Return Parameter

ER	ercd	Error Code	Error code

Error Code

Other	Error code returned by the device driver

Description

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. The processing and return from closefn must be as prompt as possible. If media ejection takes time, it is permissible to return 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.

The closefn function does not need to perform any processing with regard to multiple opening or open mode, which are handled by device management.

closefn runs as a quasi-task portion of the task that issued tk_cls_dev. When the device is closed by cleanup processing, this function is executed in the context of the cleanup function, that is, it runs as a quasi-task portion of the task that issued tk_cln_ssy.

5.3.3.2.3 execfn - Execute function

C Language Interface

```
/* Execute function (32-bit request packet, millisecond timeout) */

ER ercd = execfn (T_DEVREQ *devreq , TMO tmout , void *exinf );

/* execute function (64-bit request packet, millisecond timeout) */

ER ercd = execfn (T_DEVREQ_D *devreq_d , TMO tmout , void *exinf );

/* execute function (32-bit request packet, microsecond timeout) */

ER ercd = execfn (T_DEVREQ *devreq , TMO_U tmout_u , void *exinf );

/* execute function (64-bit request packet, microsecond timeout) */

ER ercd = execfn (T_DEVREQ_D *devreq_d , TMO_U tmout_u , void *exinf );
```

Parameter

T_DEVREQ*	devreq	Device Request Packet	Request packet (32-bit)
T_DEVREQ_D*	devreq_d	Device Request Packet	Request packet (64-bit)
TMO	tmout	Timeout	Request acceptance timeout (ms)
TMO_U	tmout_u	Timeout	Request acceptance timeout (in microseconds)
void*	exinf	Extended Information	Extended information set at device registration

Return Parameter

ER	ercd	Error Code	Error code

Error Code

Other	Error code returned by the device driver

Description

The execute function execfn is called when tk_rea_dev() or tk_wri_dev() is invoked.

Initiates the processing requested in devreq. This function initiates the requested processing only, returning to its caller without waiting for the processing to complete. The time required to initiate processing depends on the device driver; this function does not necessarily complete immediately.

When new processing cannot be accepted, this function goes to WAITING state for request acceptance. If the new request cannot be accepted within the time specified in tmout, the function times out. The TMO_POL or TMO_FEVR attribute can be specified in tmout. If the function times out, E_TMOUT is passed in the execfn return code. The request packet error parameter does not change. Timeout applies to the request acceptance, 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. In this case, the request packet is discarded. If the abort occurs 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 abort occurs, 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.

execfn runs as a quasi-task portion of the task that issued tk_rea_dev, tk_wri_dev, tk_srea_dev, or tk_swri_dev.

In a device driver for which `TDA_DEV_D` is specified as an attribute at the time of registering the device, the execute function (64-bit request packet, millisecond timeout) execfn is called when tk_rea_dev() or tk_wri_dev() is invoked. In this case, the function specification is the same as that of 32-bit request packet, millisecond timeout execfn, except that the parameter request packet is a 64-bit T_DEVREQ_D* `devreq_d`.

In a device driver for which `TDA_TMO_U` is specified as an attribute at the time of registering the device, the execute function (32-bit request packet, microsecond timeout) execfn is called when tk_rea_dev() or tk_wri_dev() is invoked. In this case, the function specification is the same as that of 32-bit request packet, millisecond timeout execfn, except that the parameter timeout specification is a microsecond TMO_U `tmout_u`.

In a device driver for which both `TDA_DEV_D` and `TDA_TMO_U` are specified as an attribute at the time of registering the device, the execute function (64-bit request packet, microsecond timeout) execfn is called when tk_rea_dev() or tk_wri_dev() is invoked. In this case, the function specification is the same as that of 32-bit request packet, millisecond timeout execfn, except that the parameter request packet is a 64-bit T_DEVREQ_D* `devreq_d` and the parameter timeout specification is a microsecond TMO_U `tmout_u`.

Difference from T-Kernel 1.0

The execute function (64-bit request packet, millisecond timeout), execute function (32-bit request packet, microsecond timeout), and execute function (64-bit request packet, microsecond timeout) were added in T-Kernel 2.0.

5.3.3.2.4 waitfn - Wait-for-completion function

C Language Interface

```
/* wait-for-completion function (32-bit request packet, millisecond timeout) */

INT creqno = waitfn (T_DEVREQ *devreq , INT nreq , TMO tmout , void *exinf );

/* wait-for-completion function (64-bit request packet, millisecond timeout) */

INT creqno = waitfn (T_DEVREQ_D *devreq_d , INT nreq , TMO tmout , void *exinf );

/* wait-for-completion function (32-bit request packet, microsecond timeout) */

INT creqno = waitfn (T_DEVREQ *devreq , INT nreq , TMO_U tmout_u , void *exinf );

/* wait-for-completion function (64-bit request packet, microsecond timeout) */

INT creqno = waitfn (T_DEVREQ_D *devreq_d , INT nreq , TMO_U tmout_u , void *exinf );
```

Parameter

T_DEVREQ*	devreq	Device Request Packet	Request packet list (32-bit)
T_DEVREQ_D*	devreq_d	Device Request Packet	Request packet list (64-bit)
INT	nreq	Number of Requests	Request packet count
TMO	tmout	Timeout	Timeout (ms)
TMO_U	tmout_u	Timeout	Timeout (in microseconds)
void*	exinf	Extended Information	Extended information set at device registration

Return Parameter

INT	creqno		Completed Request Packet Number	Completed request packet number
		or	Error Code	Error code

Error Code

Other	Error code returned by the device driver

Description

The wait-for-completion function waitfn 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 nreq must always be followed. The number of the completed request packet (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 timeout for waiting for completion is set in tmout. The TMO_POL or TMO_FEVR attribute can be specified for tmout. If the wait times out, the requested processing continues. The waitfn return code in case of timeout is

E_TMOUT. The request packet `error` parameter 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 not be completed 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 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.

I/O error and other device-related errors are stored in the request packet `error` parameter. 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`.

The abort processing when the abort function abortfn was executed during completion waiting by waitfn differs depending on whether to wait for completion of a single request (waitfn, `nreq` = 1) or multiple requests (waitfn, `nreq` > 1). When waiting for completion of a single request, the request currently processing is aborted. On the other hand, when waiting for completion of multiple requests, as a special handling, only the completion waiting by waitfn is released and the processing for the request itself is not aborted. It means that, even if the abort function abortfn is executed, the request packets' `abort` remains `FALSE` and the processing for the requests continues. E_ABORT is passed in the return code from the released waitfn.

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 than E_ABORT may be returned as appropriate based on the device properties. It is also permissible to return E_OK on the basis that the processing right up to the abort is valid. If processing completes normally to the end, E_OK is returned even if there was an abort request.

waitfn runs as a quasi-task portion of the task that issued tk_wai_dev, tk_srea_dev, or tk_swri_dev.

In a device driver for which `TDA_DEV_D` is specified as an attribute at the time of registering the device, the wait-for-completion function (64-bit request packet, millisecond timeout) waitfn is called when tk_wai_dev() is invoked. In this case, the function specification is the same as that of 32-bit request packet, millisecond timeout waitfn, except that the parameter request packet is a 64-bit T_DEVREQ_D* `devreq_d`.

In a device driver for which `TDA_TMO_U` is specified as an attribute at the time of registering the device, the wait-for-completion function (32-bit request packet, microsecond timeout) waitfn is called when tk_wai_dev() is invoked. In this case, the function specification is the same as that of 32-bit request packet, millisecond timeout waitfn, except that the parameter timeout specification is a microsecond TMO_U `tmout_u`.

In a device driver for which `TDA_DEV_D` and `TDA_TMO_U` are specified as an attribute at the time of registering the device, the wait-for-completion function (64-bit request packet, microsecond timeout) waitfn is called when tk_wai_dev() is invoked. In this case, the function specification is the same as that of 32-bit request packet, millisecond timeout waitfn, except that the parameter request packet is a 64-bit T_DEVREQ_D* `devreq_d` and the parameter timeout specification is a microsecond TMO_U `tmout_u`.

Difference from T-Kernel 1.0

The wait-for-completion function (64-bit request packet, millisecond timeout), wait-for-completion function (32-bit request packet, microsecond timeout), and wait-for-completion function (64-bit request packet, microsecond timeout) were added in T-Kernel 2.0.

5.3.3.2.5 abortfn - Abort function

C Language Interface

```
/* abort function (32-bit request packet) */
```

ER ercd = abortfn (ID tskid , T_DEVREQ *devreq , INT nreq , void *exinf);

```
/* abort function (64-bit request packet) */
```

ER ercd = abortfn (ID tskid , T_DEVREQ_D *devreq_d , INT nreq , void *exinf);

Parameter

ID	tskid	Task ID	Task ID of the task executing execfn or waitfn
T_DEVREQ*	devreq	Device Request Packet	Request packet list (32-bit)
T_DEVREQ_D*	devreq_d	Device Request Packet	Request packet list (64-bit)
INT	nreq	Number of Requests	Request packet count
void*	exinf	Extended Information	Extended information set at device registration

Return Parameter

ER	ercd	Error Code	Error code

Error Code

Other	Error code returned by the device driver

Description

The abort function abortfn is called when you want to promptly return from the currently running execute function execfn or wait-for-completion function waitfn. 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.

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 the device descriptor was processing a request, abortfn is used to abort the request being processed by the device descriptor.

`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 use 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 (waited is released), 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 request packet abort flag. 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 if 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 later. Even though processing was aborted, the request itself is not discarded until its completion has been checked by waitfn.

abortfn initiates abort processing only, returning promptly without waiting for the abort to complete.

The abortfn that is executed on a task exception runs as a quasi-task portion of the task issuing tk_ras_tex that raised the task exception. The abortfn that is executed on a device close runs as a quasi-task portion of the task that issued tk_cls_dev. When the device is closed by cleanup processing, this function is executed in the context of the cleanup function, that is, it runs as a quasi-task portion of the task that issued tk_cln_ssy.

In a device driver for which TDA_DEV_D is specified as an attribute at the time of registering the device, the abort function (64-bit request packet) abortfn is called when you want to promptly return from the currently running execute function execfn or wait-for-completion function waitfn. In this case, the function specification is the same as that of 32-bit request packet abortfn, except that the parameter request packet is a 64-bit T_DEVREQ_D* devreq_d.

Difference from T-Kernel 1.0

The abort function (64-bit request packet) was added in T-Kernel 2.0.

5.3.3.2.6 eventfn - Event function

C Language Interface

INT retcode = eventfn (INT evttyp , void *evtinf , void *exinf);

Parameter

INT	evttyp	Event Type	Driver request event type
void*	evtinf	Event Information	Information for each event type
void*	exinf	Extended Information	Extended information set at device registration

Return Parameter

INT	retcode	Return Code	Return code defined for each event type
	or	Error Code	Error code

Error Code

Other	Error code returned by the device driver

Description

When a state change occurs in the device or system which is caused by a factor other than normal device I/O processing by an application interface, requiring some processing by the device driver, a driver request event is raised and then the event function eventfn is called.

The driver request event is raised when suspending or resuming a device for power control (see tk_sus_dev) or when connecting a removable device such as USB or PC card.

For example, when the system is suspended by tk_sus_dev, the driver request event for the suspend (TDV_SUSPEND) is raised in the T-Kernel (during the tk_sus_dev processing) and the event function for each device is called with evttyp = TDV_SUSPEND. The event function called for each device performs necessary operations for suspend such as saving the state.

The following driver request events are defined.

```
#define TDV_SUSPEND   (-1)   /* suspend */
#define TDV_RESUME    (-2)   /* resume */
#define TDV_CARDEVT     1    /* PC card event */
#define TDV_USBEVT      2    /* USB event */
```

The driver request events with a negative value are called internally from the device management in the T-Kernel/SM, for suspend or resume processing.

On the other hand, the driver request events with a positive value (TDV_CARDEVT and TDV_USBEVT) are reference specifications which are not directly related to the T-Kernel operation, and raised by calling tk_evt_dev(). These driver request events are used as needed to implement a bus driver for USB, PC card, or other device.

The processing performed by the event function is defined for each event type. For suspend and resume processings, see Section 5.3.3.4, “Device Suspend/Resume Processing”.

When a device event is called by tk_evt_dev(), the eventfn return code is set transparently as the tk_evt_dev() return code.

Requests to event functions must be accepted even if another request is processed, and must be processed as quickly as possible.

The eventfn runs as a quasi-task portion of the task that issued tk_evt_dev or tk_sus_dev that caused the event.

Additional Notes

The following behaviors are assumed for PC card event or USB event.

Note that they describe implementation examples of device drivers that handle a device such as PC card or USB and are not part of the T-Kernel specification.

When a USB device is connected, a class driver should dynamically be mapped to the USB device to perform an actual I/O processing.

For example, when a storage such as USB memory is connected, a device driver for the mass storage class handles the I/O for the device, or when a USB camera is connected, a device driver for the video class handles the I/O for the device. Which device driver should be used cannot be determined until the USB device is connected.

In this case, the driver request event for the USB connection and the event function for each device driver are used in order to map a class driver to the USB device. Specifically, when the USB bus driver (USB manager) monitoring the USB ports detects a newly connected USB device, it sends the driver request event for the USB connection (`TDV_USBEVT`) to each device driver which will be candidate of the class driver and then calls the event function for each device.

The event function for each device returns whether or not it can support the newly connected USB device in response to this `TDV_USBEVT`. The USB bus driver receives the return codes and determines the mapping to the actual class driver.

The similar steps are used also for connecting PC card.

5.3.3.3 Device Event Notification

A device driver sends events that occur on each device to the specific message buffer (event notification message buffer) as device event notification messages. The event notification message buffer ID is referenced or set as an attribute data of `TDN_EVENT` for each device.

The system default event notification message buffer is used immediately after device registration. As a device is registered by tk_def_dev when a device driver is started, the system default event notification message buffer ID value is returned as this API's return parameter, the value is held in the device driver and is used as the initial value of this attribute data, `TDN_EVENT`.

The system default event notification message buffer is created at system startup. Its size and maximum message length are defined by `TDEvtMbfSz` in the system configuration information.

The message formats used in device event notification are as follows: The content and size of the event notification message vary depending on the event type.

◇Basic format of device event notification

```
typedef struct t_devevt {
        TDEvtTyp        evttyp;         /* event type */
        /* Information specific to each event type is appended here. */
} T_DEVEVT;
```

◇Format of device event notification with device ID

```
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;
```

◇Format of device event notification with extended information

```
typedef struct t_devevt_ex {
        TDEvtTyp        evttyp;         /* event type */
        ID              devid;          /* Device ID */
        UB              exdat[16];      /* Extended information */
        /* Information specific to each event type is appended here. */
} T_DEVEVT_EX;
```

The event type of a device event notification is classified as follows:

a. Basic event notification (event type: 0x0001 to 0x002F)
   Basic event notification from a device

b. System event notification (event type: 0x0030 to 0x007F)
   Event notification related to entire system such as power supply control

c. Event notification with extended information (event type: 0x0080 to 0x00FF)
   Event notification from a device with extended information

d. User-defined event notification (event type: 0x0100 to 0xFFFF)
   Notification of event that users can arbitrarily define

Typical event types are as follows: For more details on each event and other event types, see the specification related to device drivers or Section 7.1.3, “Event Type of the Device Event Notification” in Section 7.1, “Specification Related to Device Drivers to be Used as Reference”.

```
typedef enum tdevttyp {
        TDE_unknown     = 0,            /* undefined */
        TDE_MOUNT       = 0x01,         /* media insert */
        TDE_EJECT       = 0x02,         /* Eject media */
        TDE_POWEROFF    = 0x31,         /* power switch off */
        TDE_POWERLOW    = 0x32,         /* low power alarm */
        TDE_POWERFAIL   = 0x33,         /* abnormal power */
        TDE_POWERSUS    = 0x34          /* auto suspend */
} TDEvtTyp;
```

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 avoid message buffer overflow as much as possible.

Difference from T-Kernel 1.0
The description has been re-organized for message formats and event types used in the device event notification.

5.3.3.4 Device Suspend/Resume Processing

Device drivers perform suspend and resume 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.

TDV_SUSPEND
: Suspend

```
evttyp = TDV_SUSPEND
evtinf = NULL (none)
```

Suspend processing takes place as follows.

1. If there is a request being processed at the time, the device driver waits for it to complete, pauses it or aborts it. Which of these options to take depends on the device driver implementation. Since the suspension must be effected as quickly as possible, however, 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
2. New requests other than a resume event are not accepted.
3. The device power is cut off and other suspend operation is performed.

Abort should be avoided if possible because of its effects on applications. It should be used only in such cases as long input wait from a serial port, or when pause would be difficult. Normally it is best to wait for completion of a request or, if possible, choose pause (suspend and resume).

Requests arriving at the device driver in suspend state are made to wait until operation resumes, after which they are accepted for processing. If the request does not involve access to the device, however, or otherwise can be processed even during suspension, a request may be accepted without waiting for resumption.

TDV_RESUME
: Resume

```
evttyp = TDV_RESUME
evtinf = NULL (none)
```

Resume processing takes place as follows.

1. The device power is turned back on, the device states are restored and other device resume processing is performed.
2. Paused processing is resumed.
3. Accepting request is resumed.

5.3.3.5 Special Properties of Disk Devices

A disk device has a special role to play in a virtual memory system. When implementing a virtual memory system, in order to perform data transfer between memory and a disk, OS (specifically, a part to process a virtual memory in a T-Kernel Extension, etc.) needs to call a disk driver.

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 a 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, when the disk driver is waiting for a page to be read in due to the access to nonresident memory, it is possible that the OS will again request disk access to that disk driver. Even then, the disk driver must be able to execute the later OS request.

A similar case may arise in suspend processing. When access is made to nonresident memory during suspend 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, suspend 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 suspend processing a disk driver must not access nonresident memory.

Because of the above limitations, a disk driver shall not use (access) nonresident memory. It is possible, however, that the I/O buffer (`buf`) space specified with tk_rea_dev() or tk_wri_dev() can be nonresident memory since this is a memory location specified by the caller. In the case of I/O buffers, therefore, it is necessary to make the memory space resident (see LockSpace) at the time of I/O access.

5.4 Interrupt Management Functions

T-Kernel/SM interrupt management functions are functions for disabling or enabling external interrupt, retrieving interrupt disable status, controlling interrupt controller, etc.

Interrupt handling is largely hardware-dependent, different on each system, and therefore difficult to standardize. The following are given as standard specification, but it may not be possible to follow these exactly on all systems. Implementors should comply with these specifications as much as possible; but where implementation is not feasible, full compliance is not mandatory. If functions not in the standard specification are added, however, the function names must be different from those given here. In any case, DI(), EI(), and isDI() must be implemented in accordance with the standard specification.

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.

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.

DI(), EI(), and isDI() are C language macros.

5.4.1.1 DI - Disable External Interrupts

C Language Interface

```
#include <tk/tkernel.h>

DI (UINT intsts );
```

Parameter

UINT	intsts	Interrupt Status	Variable that stores the CPU external interrupt flag

Return Parameter

None

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Controls the external interrupt flag in the CPU and disables all external interrupts. Also stores the flag state in intsts before disabling interrupt.

This API is defined as a C language macro and intsts is not a pointer. Write a variable directly.

5.4.1.2 EI - Enable External Interrupt

C Language Interface

```
#include <tk/tkernel.h>
```

EI (UINT intsts);

Parameter

UINT	intsts	Interrupt Status	Variable that stores the CPU external interrupt flag

Return Parameter

None

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Controls the external interrupt flag in the CPU and reverts the flag state to intsts. That is, this API reverts the flag state to the state before disabling external interrupts by the previously executed DI(intsts).

If the state before executing DI(intsts) was the external-interrupt-enabled, the subsequent EI(intsts) enables external interrupts. On the other hand, if the state was already interrupt-disabled at the time DI(intsts) was executed, interrupt is not enabled by EI(intsts). However, if 0 is specified in intsts, the external interrupt flag in the CPU is set to the interrupt-enable state.

intsts must be either the value saved by DI() or 0. If any other value is specified, the subsequent correct behavior is not guaranteed.

5.4.1.3 isDI - Get Interrupt Disable Status

C Language Interface

```
#include <tk/tkernel.h>
```

BOOL disint = isDI (UINT intsts);

Parameter

UINT	intsts	Interrupt Status	Variable that stores the CPU external interrupt flag

Return Parameter

BOOL	disint	Interrupt Disabled Status	External interrupt disabled status

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Checks the external interrupt flag in the CPU that was stored in intsts by the previously executed DI(), and returns TRUE (a value other than 0) if the flag status is determined as the interrupt-disabled by T-Kernel/OS, or FALSE otherwise.

intsts must be the value saved by DI(). If any other value is specified, the subsequent correct behavior is not guaranteed.

Example 5.4 Sample Usage of isDI

```
void foo()
{
        UINT    intsts;

        DI(intsts);

        if ( isDI(intsts) ) {
                /* Interrupt was already disabled at the time the above DI() was called */
        } else {
                /* Interrupt was enabled at the time the above DI() was called */
        }

        EI(intsts);
}
```

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 handler numbers specified with tk_def_int(), or should allow for simple conversion to and from those numbers.

5.4.2.1 DINTNO - Convert Interrupt Vector to Interrupt Handler Number

C Language Interface

```
#include <tk/tkernel.h>

UINT dintno = DINTNO (INTVEC intvec );
```

Parameter

INTVEC	intvec	Interrupt Vector	Interrupt vector

Return Parameter

UINT	dintno	Interrupt Handler Number	Interrupt handler number

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Converts an interrupt vector to the corresponding interrupt handler number.

5.4.2.2 EnableInt - Enable Interrupts

C Language Interface

```
#include <tk/tkernel.h>
```

void EnableInt (INTVEC intvec);
void EnableInt (INTVEC intvec , INT level);

Parameter

INTVEC	intvec	Interrupt Vector	Interrupt vector
INT	level	Interrupt Priority Level	Interrupt priority level

Return Parameter

None

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Enables the interrupt specified in `intvec`. 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.

Either methods with or without `level` shall be provided.

5.4.2.3 DisableInt - Disable Interrupts

C Language Interface

```
#include <tk/tkernel.h>
```

void DisableInt (INTVEC intvec);

Parameter

INTVEC	intvec	Interrupt Vector	Interrupt vector

Return Parameter

None

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Disables the interrupt specified in intvec. Generally, interrupts raised while the 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 raised during interrupt-disabled-state.

5.4.2.4 ClearInt - Clear Interrupt

C Language Interface

```
#include <tk/tkernel.h>
```

void ClearInt (INTVEC intvec);

Parameter

INTVEC	intvec	Interrupt Vector	Interrupt vector

Return Parameter

None

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Clears interrupts raised for intvec, if any.

5.4.2.5 EndOfInt - Issue EOI to Interrupt Controller

C Language Interface

```
#include <tk/tkernel.h>
```

void EndOfInt (INTVEC intvec);

Parameter

INTVEC	intvec	Interrupt Vector	Interrupt vector

Return Parameter

None

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

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.

5.4.2.6 CheckInt - Check Interrupt

C Language Interface

```
#include <tk/tkernel.h>

BOOL rasint = CheckInt (INTVEC intvec );
```

Parameter

INTVEC	intvec	Interrupt Vector	Interrupt vector

Return Parameter

BOOL	rasint	Interrupt Raised Status	External interrupt raised status

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Checks whether an interrupt for intvec has been raised. If an interrupt for intvec has been raised, it returns TRUE (value other than 0), else returns FALSE.

5.4.2.7 SetIntMode - Set Interrupt Mode

C Language Interface

```
#include <tk/tkernel.h>
```

void SetIntMode (INTVEC intvec , UINT mode);

Parameter

INTVEC	intvec	Interrupt Vector	Interrupt vector
UINT	mode	Mode	Interrupt mode

Return Parameter

None

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Sets the interrupt specified in intvec for the mode specified in mode.

The settable modes and how to specify mode are implementation-dependent. The following is an example of settable modes:

```
mode := (IM_LEVEL || IM_EDGE) | (IM_HI || IM_LOW)
```

```
#define IM_LEVEL        0x0002          /* Level trigger */
#define IM_EDGE         0x0000          /* Edge trigger */
#define IM_HI           0x0000          /* H level/Interrupt at rising edge */
#define IM_LOW          0x0001          /* L level/Interrupt at falling edge */
```

If invalid mode is specified, the subsequent correct behavior is not guaranteed.

Difference from T-Kernel 1.0

This API was added in T-Kernel 2.0.

5.5 I/O Port Access Support Functions

I/O port access support functions support accesses or operations to the I/O devices. These include functions that read from or write to the I/O port of the specified address using the unit of byte or word, and a function that realizes a wait for a short time (micro wait) which is used for I/O device operations.

I/O port access support functions are provided as library functions or C language macros. These can be called from a task-independent portion or while task dispatching and interrupts are disabled.

5.5.1 I/O Port Access

In a system with separate I/O space and memory space, I/O port access functions access I/O space. In a system with memory-mapped I/O only, I/O port access functions access memory space. Using these functions will improve software portability and readability even in a memory-mapped I/O system.

5.5.1.1 out_b - Write to I/O Port (In Unit of Byte)

C Language Interface

```
#include <tk/tkernel.h>
```

void out_b (INT port , UB data);

Parameter

INT	port	I/O Port Address	I/O port address
UB	data	Write Data	Data to be written (in unit of byte)

Return Parameter

None

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Writes `data` in byte (8-bit) to the I/O port pointed by the address `port`.

5.5.1.2 out_h - Write to I/O Port (In Unit of Half-word)

C Language Interface

```
#include <tk/tkernel.h>
```

void out_h (INT port , UH data);

Parameter

INT	port	I/O Port Address	I/O port address
UH	data	Write Data	Data to be written (in unit of half-word)

Return Parameter

None

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Writes `data` in a half-word (16-bit) to the I/O port pointed by the address `port`.

5.5.1.3 out_w - Write to I/O Port (In Unit of Word)

C Language Interface

```
#include <tk/tkernel.h>
```

void out_w (INT port , UW data);

Parameter

INT	port	I/O Port Address	I/O port address
UW	data	Write Data	Data to be written (in unit of word)

Return Parameter

None

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Writes data in a word (32-bit) to the I/O port pointed by the address port.

5.5.1.4 out_d - Write to I/O Port (In Unit of Double-word)

C Language Interface

```
#include <tk/tkernel.h>
```

void out_d (INT port , UD data);

Parameter

INT	port	I/O Port Address	I/O port address
UD	data	Write Data	Data to be written (in unit of double-word)

Return Parameter

None

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Writes `data` in a double-word (64-bit) to the I/O port pointed by the address `port`.

Note that, in a system where I/O port cannot be accessed in double-word (64-bit) due to hardware constraint, `data` is separated into shorter units than double-word (64-bit) before they are written.

Rationale for the Specification

There are many systems where I/O port cannot be accessed in double-word (64-bit) due to hardware constraint such as 32-bit or less I/O data bus. In such systems, the strict specification of out_d() and in_d() cannot be implemented; that is, they cannot process `data` in one chunk of the specified bit width. In terms of the original purpose of this API, it is preferable not to implement the out_d() and in_d() or return an error at runtime. However, it is not practical to detect an error by determining the bus configuration at runtime, and it is often harmless to separate 64-bit data into 32-bit or narrower units before writing.

This is why the specification of out_d() and in_d() allow for the case where 64-bit data cannot be processed in one chunk. Therefore, whether out_d() and in_d() support the block access to 64-bit I/O port or not is implementation-dependent. If the block access to 64-bit I/O port is needed, the system hardware configuration and handling of out_d() and in_d() should be checked.

Difference from T-Kernel 1.0

This API was added in T-Kernel 2.0.

5.5.1.5 in_b - Read from I/O Port (In Unit of Byte)

C Language Interface

```
#include <tk/tkernel.h>
```

UB data = in_b (INT port);

Parameter

INT	port	I/O Port Address	I/O port address

Return Parameter

UB	data	Read Data	Data to be read (in unit of byte)

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Reads data in a byte (8-bit) from the I/O port pointed by the address `port` and returns it in the return parameter `data`.

5.5.1.6 in_h - Read from I/O Port (In Unit of Half-word)

C Language Interface

```
#include <tk/tkernel.h>

UH data = in_h (INT port );
```

Parameter

INT	port	I/O Port Address	I/O port address

Return Parameter

UH	data	Read Data	Data to be read (in unit of half-word)

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Reads data in a half-word (16-bit) from the I/O port pointed by the address `port` and returns it in the return parameter `data`.

5.5.1.7 in_w - Read from I/O Port (In Unit of Word)

C Language Interface

```
#include <tk/tkernel.h>
```

UW data = in_w (INT port);

Parameter

INT	port	I/O Port Address	I/O port address

Return Parameter

UW	data	Read Data	Data to be read (in unit of word)

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Reads data in a word (32-bit) from the I/O port pointed by the address `port` and returns it in the return parameter `data`.

5.5.1.8 in_d - Read from I/O Port (In Unit of Double-word)

C Language Interface

```
#include <tk/tkernel.h>

UD data = in_d (INT port );
```

Parameter

INT	`port`	I/O Port Address	I/O port address

Return Parameter

UD	`data`	Read Data	Data to be read (in unit of double-word)

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Reads data in a double-word (64-bit) from the I/O port pointed by the address `port` and returns it in the return parameter `data`.

Note that, in a system where I/O port cannot be accessed in one chunk of double-word (64-bit) due to hardware constraint, data is separated into shorter units than double-word (64-bit) before reading.

Rationale for the Specification

See Section 5.5.1.4, “out_d - Write to I/O Port (In Unit of Double-word)”.

Difference from T-Kernel 1.0

This API was added in T-Kernel 2.0.

5.5.2 Micro Wait

5.5.2.1 WaitUsec - Micro Wait (in Microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

void WaitUsec (UINT usec);

Parameter

UINT	usec	Micro Seconds	Wait time (microseconds)

Return Parameter

None

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Performs a micro wait for the specified interval (in microseconds).

This wait is usually implemented as a busy loop. This means that the micro wait occurs in the task RUNNING state rather than WAITING state.

The micro wait is easily influenced by the runtime environment, such as execution in RAM, execution in ROM, memory cache on or off, etc. The wait time is therefore not very accurate.

5.5.2.2 WaitNsec - Micro Wait (in Nanoseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

void WaitNsec (UINT nsec);

Parameter

UINT	nsec	Nanoseconds	Wait time (nanoseconds)

Return Parameter

None

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Performs a micro wait for the specified interval (in nanoseconds).

This wait is usually implemented as a busy loop. This means that the micro wait occurs in the task RUNNING state rather than WAITING state.

The micro wait is easily influenced by the runtime environment, such as execution in RAM, execution in ROM, memory cache on or off, etc. The wait time is therefore not very accurate.

5.6 Power Management Functions

Power management functions are used to realize system power saving. Power management functions are called as a callback type function from within T-Kernel/OS.

Though low_pow() off_pow() exist as part of APIs that are defined in the power management function, they are reference specification and should be used only internally inside the T-Kernel. Since device drivers, middleware, and applications do not call these APIs directly, it is allowed to modify the functions or their APIs in the original specification to realize more advanced power management function. If, however, the functions implemented have only the equivalent or similar performance as the APIs being defined as a reference specification here, it is preferable to follow this reference specification in order to enhance the program reusability.

Calling method of APIs for these functions is also implementation-dependent. Simple system calls are possible, as is the use of a trap. These functions may be provided in programs other than the T-Kernel. Use of an extended SVC or other means that makes use of T-Kernel function is not possible, however.

5.6.1 low_pow - Move System to Low-power Mode

C Language Interface

```
void low_pow ( void );
```

Parameter

None

Return Parameter

None

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
NO	NO	NO

Description

Called from the T-Kernel task dispatcher to move the CPU and its associated hardware to the low-power mode.

After moving CPU to the low-power mode, low_pow() waits for an external interrupt. When an external interrupt occurs, low_pow() moves the CPU and its associated hardware back to the normal mode (non low-power mode) and then returns to the caller of it.

The detailed processing procedure for low_pow() is as follows:

1. Move CPU to the low-power mode. For example, lower the clock frequency.
2. Stop CPU, waiting for an external interrupt. For example, execute such a CPU instruction.
3. Resume CPU after an external interrupt (by hardware).
4. Move the CPU back to the normal mode. For example, restore the normal clock frequency.
5. Return to the caller. The actual caller is the dispatcher in T-Kernel.

When implementing low_pow(), the following points need to be noted:

- This function is called in interrupts disabled state.
- Interrupts must not be enabled.
- Since the processing speed affects the speed of response to an interrupt, it should be as fast as possible.

Additional Notes

The task dispatcher calls low_pow() to lower the power consumption when it has no tasks to be executed.

5.6.2 off_pow - Move System to Suspend State

C Language Interface

```
void off_pow ( void );
```

Parameter

None

Return Parameter

None

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
NO	NO	NO

Description

Called from T-Kernel during the processing of tk_set_pow() with `powmode` = `TPW_DOSUSPEND` to move the CPU and its associated hardware to the suspend state (power off state).

After moving the hardware to the suspend state, off_pow() waits for a resume factor (power on, etc.). When a resume factor occurs, off_pow() releases the suspend state and then returns to the caller of it.

The detailed processing procedure for off_pow() is as follows:

1. Move CPU to the suspend state and wait for a resume factor. For example, stop the clock.
2. Resume CPU on the occurrence of a resume factor (by hardware).
3. Move CPU or other hardware back to the normal state, if necessary. Release the suspend state.(may be processed by hardware together with the previous step)
4. Return to the caller. The actual caller is the processing part of tk_set_pow() in T-Kernel.

When implementing off_pow(), the following points need to be noted:

- This function is called in interrupts disabled state.
- Interrupts must not be enabled.

Note that the device drivers perform the suspending and resuming of peripherals and other devices. For more details, see the description of tk_sus_dev().

5.7 System Configuration Information Management Functions

System configuration information management functions maintain and manage various information related to system configuration.

A part of system configuration information including the information on the maximum number of tasks, timer interrupt intervals, etc. are defined as the standard definition. Other than these, any information arbitrarily defined in applications, subsystems, or device drivers can be used by adding it to the system configuration information.

The format of system configuration information consists of a name and defined data as a pair.

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 '_' (underscore).

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 (in character code).

Example 5.5 Example of Format of System Configuration Information

Name	Defined Data
SysVer	1 0
SysName	T-Kernel Version 1.00

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.

System configuration information can be retrieved by tk_get_cfn and tk_get_cfs.

However, system configuration information cannot be added or changed during system execution.

5.7.1 System Configuration Information Acquisition

There are tk_get_cfn and tk_get_cfs as extended SVCs to retrieve system configuration information. These are callable from applications, subsystems, device drivers, etc. and are also used internally in the T-Kernel. Usage inside T-Kernel does not have to go through extended SVC; this choice is implementation-dependent.

5.7.1.1 tk_get_cfn - Get Numbers

C Language Interface

```
#include <tk/tkernel.h>
```

INT ct = tk_get_cfn (CONST UB *name , INT *val , INT max);

Parameter

CONST UB*	name	Name	Name
INT*	val	Value	Array storing numbers
INT	max	Maximum Count	Number of elements in val array

Return Parameter

INT	ct	Defined Numeric Information Count	Number of defined numeric information
	or	Error Code	Error code

Error Codes

E_NOEXS	No information is defined for the name specified in the name parameter

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Gets numeric information from system configuration information. This function gets up to max items of numerical information defined for the name specified in the name parameter and stores the acquired information in val. The number of defined numeric 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 defined numeric values can be found out without actually storing them in val.

E_NOEXS is returned if no information is defined for the name specified in the name parameter. The behavior if the information defined for name is a character string is indeterminate.

This function can be invoked from any protection level, without being limited to the protection level from which T-Kernel/OS system call can be invoked.

5.7.1.2 tk_get_cfs - Get Character String

C Language Interface

```
#include <tk/tkernel.h>
```

INT rlen = tk_get_cfs (CONST UB *name , UB *buf , INT max);

Parameter

CONST UB*	name	Name	Name
UB*	buf	Buffer	Array storing character string
INT	max	Maximum Length	Maximum size of buf (in bytes)

Return Parameter

INT	rlen	Size of Defined Character String Information	Size of defined character string information (in bytes)
	or	Error Code	Error code

Error Codes

E_NOEXS	No information is defined for the name specified in the name parameter

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Gets character string information from system configuration information. This function gets up to max characters of character string information defined for 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 for the name specified in the name parameter. The behavior if the information defined for name is a numeric string is indeterminate.

This function can be invoked from any protection level, without being limited to the protection level from which T-Kernel/OS system call can be invoked.

5.7.2 Standard System Configuration Information

The following information is defined as standard system configuration information. A standard information name is prefixed by T.

character string	Summary description
N	Numeric string information
S	Character string information

- Product information

character string	Name of standard definition	Summary description
S	TSysName	System name (product name)

- Maximum number of objects

character string	Name of standard definition	Summary description
N	TMaxTskId	Maximum number of tasks
N	TMaxSemId	Maximum number of semaphores
N	TMaxFlgId	Maximum number of event flags
N	TMaxMbxId	Maximum number of mailboxes
N	TMaxMtxId	Maximum number of mutexes
N	TMaxMbfId	Maximum number of message buffers
N	TMaxPorId	Maximum number of rendezvous ports
N	TMaxMpfId	Maximum number of fixed-size memory pools
N	TMaxMplId	Maximum number of variable-size memory pools
N	TMaxCycId	Maximum number of cyclic handlers
N	TMaxAlmId	Maximum number of alarm handlers
N	TMaxResId	Maximum number of resource groups
N	TMaxSsyId	Maximum number of subsystems
N	TMaxSsyPri	Maximum number of subsystem priorities

- Other

character string	Name of standard definition	Summary description
N	TSysStkSz	Default system stack size (in bytes)
N	TSVCLimit	Lowest protection level for system call invoking
N	TTimPeriod	Timer interrupt interval (in milliseconds)Timer interrupt interval (in microseconds)

The actual length of timer interrupt interval is a sum of time in milliseconds and time in microseconds. The interval in microseconds is assumed to be 0 when omitted.

For example, when timer interrupt interval should be 5 milliseconds, describe as "TTimPeriod 5" or "TTimPeriod 0 5000". When timer interrupt interval should be 1.5 milliseconds (1,500 microseconds), describe as "TTimPeriod 1 500" or "TTimPeriod 0 1500".

- device management function

character string	Name of standard definition	Summary description
N	TMaxRegDev	Maximum number of device registrations
N	TMaxOpnDev	Maximum device open count
N	TMaxReqDev	Maximum number of 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.

Example 5.6 Example of Storage Order of More than One Numeric Value

```
tk_get_cfn("TDEvtMbfSz", val, 2)
```

val[0] = Event notification message buffer size
val[1] = Maximum event notification message length

Difference from T-Kernel 1.0
Setting information in microseconds is added as the second element of TTimPeriod.

5.8 Memory Cache Control Functions

Memory cache control functions perform a cache control or mode setting.

The approach of cache control in T-Kernel are as follows:

Basically, even if application and device driver programs are created without paying attention to the existence of cache, the appropriate cache control should be automatically performed during their execution. Especially, in consideration of program portability, functions with strong dependency on system including cache are better to be handled separately from application programs wherever possible. For this reason, it is the policy of individual systems based on T-Kernel to make the T-Kernel itself control the cache automatically.

Specifically, T-Kernel sets the cache so that it is turned on for space like memory to store usual programs or data, and off for space such as I/O. For this reason, ordinary application programs do not need to explicitly call a function for cache control. Appropriate cache control is automatically performed even if cache control is not explicitly performed from the program.

However, the cache control by T-Kernel only (cache control by default setting) may not be enough for particular situations. For example, for I/O processing with DMA transfer or using memory space outside the kernel management, explicit cache control may be required. When executing a program by dynamically loading or generating (compiling) it, such cache control may be required so that data cache and instruction cache are appropriately synchronized. Memory cache control functions are assumed to be used in these situations.

Difference from T-Kernel 1.0
These functions were added in T-Kernel 2.0.

5.8.1 SetCacheMode - Set Cache Mode

C Language Interface

```
#include <tk/tkernel.h>
```

INT rlen = SetCacheMode (void *addr , INT len , UINT mode);

Parameter

void*	addr	Start Address	Start address
INT	len	Length	memory area size (in bytes)
UINT	mode	Mode	Cache mode

Return Parameter

INT	rlen		Result Length	Size of the area for which the cache mode was set (in bytes)
		or	Error Code	Error code

Error Code

E_OK	Normal completion
E_PAR	Parameter error (addr, len, or mode is invalid or cannot be used)
E_NOSPT	Unsupported function (function specified in mode is unsupported)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Sets the cache mode for a memory area. Specifically, performs the setting specified in mode for the cache of the len bytes memory area from the address addr. The memory cache mode is set in page units.

```
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 Applies the cache setting only for the contiguous physical address space
                   ...
                /* Implementation-dependent mode may be added */
```

Specify CM_OFF in mode to flush (writes back) the cache, invalidate it, and turn it off.

Specify CM_WT in mode to flush the cache and then set the write through cache mode.

Specify CM_WB in mode to set the write back cache mode. In this case, whether or not to flush the cache is implementation-dependent.

Specify CM_CONT in mode to apply the cache mode setting only for the contiguous physical address space area from addr. If a non-contiguous physical address or a paged out area exists within the specified area that corresponds to the specified logical memory space area, the processing is aborted immediately before the non-contiguous physical address and the size of the processed area is returned. If CM_CONT is not specified, the cache is processed for the entire specified area and the size of the processed area is returned.

Some or all of the cache mode settings may be unusable depending on CPU or implementation. If an unusable mode is specified, E_NOSPT is returned without any processing.

len must be 1 or more. If a value of 0 or less is specified, the error code E_PAR is returned.

Additional Notes

Because the cache mode setting is performed in page units, the start address of the page including addr and subsequent addresses is taken as the setting target when addr is not on the page border. Note that unintended cache access may occur to adjacent area when using this API. The page size is implementation-dependent and can be obtained using GetSpaceInfo.

When you want more detailed cache mode settings depending on the hardware configuration or the cache function of CPU, add and use an implementation-dependent mode. For example, NORMAL CACHE OFF (Weakly Order), DEVICE CACHE OFF (Weakly Order), STRONG ORDER, or other cache mode may be specified.

When an unavailable mode is specified, it is implementation-dependent whether to generate an error as E_NOSPT or E_PAR.

Difference from T-Kernel 1.0

This API was added in T-Kernel 2.0.

In T-Kernel 1.0, CnvPhysicalAddr was supported to perform the DMA transfer using the physical address. This single API performs the following three operations: (a) convert the logical address to the physical address, (b) write back the cache as preprocessing of the DMA transfer, and (c) disable the cache of the DMA transfer buffer space. However, some of these three operations are often unnecessary and it may be more efficient to invoke only the necessary operations. In addition, some device drivers for other OSes assume that the operation (a), (b), or (c) are provided separately, and it is more convenient to invoke the operations (a), (b), or (c) separately when you want to port them to T-Kernel. Therefore, in T-Kernel 2.0, these three operations performed in CnvPhysicalAddr are separated into three new APIs to get address space information (GetSpaceInfo), set cache mode (SetCacheMode), and control cache (ControlCache).

5.8.2 ControlCache - Control Cache

C Language Interface

```
#include <tk/tkernel.h>
```

INT rlen = ControlCache (void *addr , INT len , UINT mode);

Parameter

void*	addr	Start Address	Start address
INT	len	Length	Memory area size (in bytes)
UINT	mode	Mode	Control mode

Return Parameter

INT	rlen		Result Length	Size of the area for which the cache mode was set (in bytes)
		or	Error Code	Error code

Error Code

E_OK	Normal completion
E_PAR	Parameter error (invalid addr, len or mode)
E_NOSPT	Unsupported function (function specified in mode is unsupported)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Control the cache (flush or invalidate) of a memory area. Specifically, performs the control specified in mode for the cache of the len bytes memory area from the logical address addr.

```
mode := (CC_FLUSH | CC_INVALIDATE)
                CC_FLUSH        Flush (write back) cache
                CC_INVALIDATE   Invalidate cache
                     ...
                /* Implementation-dependent mode values may be added */
```

Both CC_FLUSH and CC_INVALIDATE can be set at the same time. This combination flushes the cache and then invalidates it.

If the processing is successful, the size of the processed space is returned. If a paged out area exists within the specified space, the processing is aborted immediately before it and the size of the processed space is returned.

A range that spans areas with different cache modes or attributes must not be specified. For example, a range that spans areas with cache on and cache off, task space and task shared space, or areas with different

protection levels must not be specified. If such a range is specified, the subsequent correct behavior is not guaranteed.

The detail of the function varies depending on CPU, hardware, or implementation because the cache control depends heavily on the hardware. The cache control is basically applied on the specified area using the specified mode, but it may affect more area including the specified area. For example, there are the following cases:

- Only the exactly specified range is not always controlled (flushed or invalidated). An area including the specified range is controlled, but it is also possible to flush or invalidate the cache for other areas (for example, entire memory) depending on CPU, hardware, or implementation.
- Normally, no operation is performed when a cache-off area is specified. Even in this case, it is possible to flush or invalidate the cache for areas other than the specified range.(always flush the entire space, etc.)
- No operation is performed in a system without cache.

Generally, the cache control is performed in cache line size units. For this reason, note that unintended cache access may occur to adjacent area when using this API. The cache line size is implementation-dependent and can be obtained using GetSpaceInfo.

Difference from T-Kernel 1.0

This API was added in T-Kernel 2.0.

5.9 Physical Timer Functions

Physical timer functions are useful in the system equipped with more than one hardware timer when processing should be performed based on smaller unit of elapsed time than the timer interrupt interval (TTimPeriod).

A physical timer means a hardware counter that is monotonically incremented by one from 0 at a constant time interval. When a count value reaches a certain value (upper limit) specified for each physical timer, the handler (physical timer handler) specified for each physical timer is started and the count value is reset to 0.

More than one physical timer can be used depending on the number of hardware timers available in the system. The number of available physical timers is implementation-dependent. In the usual T-Kernel implementation, one hardware timer is used to realize the time management functions. Therefore it is assumed that remaining hardware timers are used for the physical timers.

Positive integer of ascending order like 1, 2, ... is used as a physical timer number. For example, when there are four hardware timers, as one of them is used for the T-Kernel time management functions, remaining three hardware timers are available with physical timer numbers assigned as 1, 2, and 3, respectively.

The T-Kernel/SM physical timer functions do not manage coordination between an individual physical timer and tasks that use the timer. If more than one task share one physical timer, coordination like mutual exclusion control must be performed on the application side.

Additional Notes

For the T-Kernel time management functions, the kernel starts alarm handler or cyclic handler, processes timeout, and processes these requests, all in the handler that is started on the time interval specified by "timer interrupt interval" (TTimPeriod) in Section 5.7.2, “Standard System Configuration Information”. On the other hand, the physical timer functions only standardize the primitive functions such as setting a hardware timer, reading a count value, and triggering interrupt. They do not perform multiple processings like the T-Kernel time management functions do. Based on this observation, the physical timer functions carry the name of "physical timer" since they have lower abstraction level than conventional time management functions, and are closer to hardware layer.

Due to the above positioning, the physical timer functions are made to be as simple as possible and limited to a small specification, and are assumed to be realized by library functions which have small overhead. This policy is reflected in the specification of using the statically fixed physical timer numbers rather than dynamical ID numbers, and the specification of never performing the management of mapping with the requesting task or the requests from more than one task.

Physical timer functions are implemented by standardizing APIs that operate the timer (counter) device. However, the timer devices have direct relation with time related behaviors such as calling interrupt handler based on a small elapsed time, making such devises more closely connected with the kernel than other devices (storage and communication). For this reason, the physical timer is provided as more generic function by standardizing its specification as a part of the T-Kernel/SM instead of standardizing it as part of device driver specification.

Since the physical timer functions belong to the T-Kernel/SM, the T-Kernel/SM [Overall Note and Supplement] is applied.

Hardware timer counter used as a physical timer is assumed to be 32-bit or less. Therefore, 32-bit UW is used for the data type that represents the count values or upper limits. In the future, 64-bit functions can be added.

Rationale for the Specification

In the T-Kernel 2.0, the time management functions are enhanced, and the physical timer functions have been introduced in order to make effective use of multiple hardware timers implemented on the recent embedded microcomputers or SoC (System on a Chip) and enhance the portability of programs that operate these timers.

Difference from T-Kernel 1.0

These functions were added in T-Kernel 2.0.

5.9.1 Use Case of Physical Timer

Examples of effective use of physical timer functions are as follows:

(a)Example of processing to be realized

Assume that there are a cyclic processing X to be run every 2,500 microseconds and a cyclic processing Y to be run every 1,800 microseconds. Physical timers can achieve this efficiently.

(b)Implementation with physical timer functions

Two physical timers are used, and one is set to start a physical timer handler every 2,500 microseconds.

For example, if the physical timer clock frequency is 10 MHz, as 1 clock corresponds to 0.1 microseconds (= 100 nanoseconds), set a physical timer upper limit (`limit`) to 24,999 (= 25,000 - 1) to make the physical timer handler start when the count value is changed from 24,999 to 0.

As this is a cyclic processing, `mode` of StartPhysicalTimer should be set to `TA_CYC_PTMR`.

Processing X is performed within this physical timer handler.

Similarly using another physical timer, the physical timer handler is set to start every 1,800 microseconds to perform the processing Y within this physical timer handler.

The timer interrupt interval (`TTimPeriod`) used by the T-Kernel time management functions can be left as the default value (10 milliseconds) since it has no relationship with the physical timer functions.

(c)Implementation without physical timer functions

Instead of the physical timer handler, the T-Kernel 2.0 system call (tk_cre_cyc_u) that can specify time in microseconds is used to define the cyclic handler to start it every 2,500 microseconds to perform the processing X within this cyclic handler. Similarly, a cyclic handler is defined to start it every 1,800 microseconds to perform the processing Y within this cyclic handler.

However, in this case, the timer interrupt interval must be set with small enough interval so that the time of every 2,500 microseconds and every 1,800 microseconds can be processed precisely. Specifically, both processing every 2,500 microseconds and processing every 1,800 microseconds can be achieved with almost exact timing by using the timer interrupt interval of 100 microseconds which is a common divisor of 2,500 microseconds and 1,800 microseconds.

With the method (b) which uses the physical timer functions, the timer interrupt interval can be left as the default value (every 10 milliseconds) since the T-Kernel time management functions are not used. Interrupts by the physical timer will occur every 2,500 and 1,800 microseconds, from which the physical timer handler is called to perform the processing X and Processing Y. No unnecessary interrupt related to timer will occur other than these.

On the other hand, for the method of (c) which does not use a physical timer, because the timer interrupt interval must be shortened, the overhead increases accordingly as the number of timer interrupts increases. For example, when comparing (b) and (c) in terms of the number of timer related interrupts that occur in 10 milliseconds period, (b) will have a total interrupt number of 10; 1 (= 10 milliseconds/10 milliseconds) for time management functions, 4 (= 10 milliseconds/2,500 microseconds) as physical timer interrupt for processing X, and 5 (= 10 milliseconds/1,800 microseconds) as physical timer interrupt for processing Y. For (c), timer interrupt number is 100 (10 milliseconds/100 microseconds) for time management functions. This is a trade-off situation with the accuracy of time. The smaller timer interval may be required depending on the difference between cycles or phases of processing X and processing Y, resulting in even larger overhead. In these cases, the physical timer functions are clearly effective.

However, the physical timer functions are highly effective only when the number of processings that depend on time is small and statically fixed, and enough number of hardware timers exist for them. Because the physical timer functions are, as its name shows, subject to the constraints of physical hardware resources, physical timer functions cannot be used effectively when the number of hardware timers is too small. Additionally, it will experience difficulty with the case where the number of time-dependent processings dynamically increases. In these cases, using the conventional time management functions such as the cyclic handler and alarm handler will achieve more flexible handling.

Though the application area of physical timer functions and time management functions in microseconds may overlap, they have different characteristics shown above. Therefore, it is recommended to use appropriate one depending on the hardware configuration and applications. The physical timer functions have been added for this reason.

5.9.2 StartPhysicalTimer - Start Physical Timer

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = StartPhysicalTimer (UINT ptmrno , UW limit , UINT mode);

Parameter

UINT	ptmrno	Physical Timer Number	Physical timer number
UW	limit	Limit	Upper limit
UINT	mode	Mode	Operation mode

Return Parameter

ER	ercd	Error Code	Error code

Error Code

E_OK	Normal completion
E_PAR	Parameter error (ptmrno, limit, or mode is invalid or cannot be used)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Sets the count value of the physical timer specified by ptmrno to 0, and then starts counting. After this function is executed, the count value is incremented by one at a constant time interval that is the inverse of the timer clock frequency.

limit specifies the upper limit of the count value. When a time period equal to the inverse of the clock frequency has elapsed after the count value reaches the upper limit, the count value is reset to 0. At that timing, if a physical timer handler is defined for this physical timer, that handler will be started. The duration between when the counting is started by StartPhysicalTimer() call and when the counter is reset to zero is (inverse of timer clock frequency) x (upper limit + 1).

If limit is set to 0, an E_PAR error will occur.

mode specifies the following modes:

TA_ALM_PTMR	0	The counting is stopped when the count value is reset to 0 from the upper limit value. Afterward, the count value remains as 0.
TA_CYC_PTMR	1	The count value starts to increase again, after it is reset to 0 from the upper limit value. Therefore, the cycle of increasing and resetting the count value repeats periodically.

Difference from T-Kernel 1.0

This API was added in T-Kernel 2.0.

5.9.3 StopPhysicalTimer - Stop Physical Timer

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = StopPhysicalTimer (UINT ptmrno);

Parameter

UINT	ptmrno	Physical Timer Number	Physical timer number

Return Parameter

ER	ercd	Error Code	Error code

Error Code

E_OK	Normal completion
E_PAR	Parameter error (ptmrno is invalid or cannot be used)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Stops the counting operation of the physical timer specified by ptmrno.

After executing this function, the last count value of the physical timer is retained. Therefore, if GetPhysicalTimerCount is executed after this function is executed, that function will return the physical timer count value just before this function is executed.

Executing this function for the physical timer that has already stopped counting does nothing. It does not generate any error.

Additional Notes

If the physical timer that is no longer used is kept running, it may not adversely affect the program operation, but clock signals will be used unnecessarily, which may not be desirable in terms of electric power saving. So, it is recommended to stop the physical timer no longer used by executing this function.

Use of this function is effective for the case TA_CYC_PTMR is specified for the physical timer and its use is ended. If TA_ALM_PTMR is specified as the mode, the physical timer automatically stopped counting after the count value is reset to 0 from the upper limit value, which results in the same state as that after this function being executed. In this case, it is not necessary to issue this function additionally. Issuing this function does not cause any problem, but nothing is changed.

Difference from T-Kernel 1.0

This API was added in T-Kernel 2.0.

5.9.4 GetPhysicalTimerCount - Get Physical Timer Count

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = GetPhysicalTimerCount (UINT ptmrno , UW *p_count);

Parameter

UINT	ptmrno	Physical Timer Number	Physical timer number
UW*	p_count	Pointer to Physical Timer Count	Pointer to the area to return the current physical timer count

Return Parameter

ER	ercd	Error Code	Error code
UW	count	Physical Timer Count	Current count value

Error Code

E_OK	Normal completion
E_PAR	Parameter error (ptmrno is invalid or cannot be used)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Gets the current count value of the physical timer specified by ptmrno, and returns it as the return parameter count.

Difference from T-Kernel 1.0

This API was added in T-Kernel 2.0.

5.9.5 DefinePhysicalTimerHandler - Define Physical Timer Handler

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = DefinePhysicalTimerHandler (UINT ptmrno , CONST T_DPTMR *pk_dptmr);

Parameter

UINT	ptmrno	Physical Timer Number	Physical timer number
CONST T_DPTMR*	pk_dptmr	Packet to Define Physical Timer Handler	Physical timer handler definition information

pk_dptmr Detail

void*	exinf	Extended Information	Extended information
ATR	ptmratr	Physical Timer Attribute	Physical timer handler attribute (TA_ASM || TA_HLNG)
FP	ptmrhdr	Physical Timer Handler Address	Physical timer handler address

Return Parameter

ER	ercd	Error Code	Error code

Error Code

E_OK	Normal completion
E_NOMEM	Insufficient memory (memory for control block cannot be allocated)
E_RSATR	Reserved attribute (ptmratr is invalid or cannot be used)
E_PAR	Parameter error (ptmrno, pk_dptmr, or ptmrhdr is invalid or cannot be used, or the physical timer handler for ptmrno cannot be defined)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

If pk_dptmr is not NULL, this function defines the physical timer handler for the physical timer specified by ptmrno. The physical timer handler is a handler running as a task-independent portion, and is started when the physical timer count is reset to 0 from the upper limit value specified by limit of StartPhysicalTimer.

The programming format of physical timer handler is similar to that of cyclic handler or alarm handler. This means that if the TA_HLNG attribute is specified, the physical timer handler is started via a high-level language support routine and terminated by a return from the function. If the TA_ASM attribute is specified, the physical timer handler format is implementation-dependent. Regardless of which attribute is specified, exinf is passed as a startup parameter of physical timer handler.

If pk_dptmr is NULL, this function cancels the definition of the physical timer handler for the physical timer specified by ptmrno. The physical timer handlers for all the physical timers are undefined right after the system startup.

If the physical timer handler for the physical timer specified by ptmrno cannot be defined (if the pk_rptmr->defhdr in GetPhysicalTimerConfig returns FALSE), the E_PAR error occurs. If the physical timer specified by ptmrno does not exist or cannot be used, the E_PAR error also occurs.

Additional Notes

In an implementation, the interrupt handler to realize the physical timer function should be defined within T-Kernel/SM, and set to be started when the physical timer count is reset to 0 from the upper limit value. Within this interrupt handler, call the physical timer handler defined by this function, and perform the processing related to the physical timer implementation (such as one related to TA_ALM_PTMR and TA_CYC_PTMR).

Difference from T-Kernel 1.0

This API was added in T-Kernel 2.0.

5.9.6 GetPhysicalTimerConfig - Get Physical Timer Configuration Information

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = GetPhysicalTimerConfig (UINT ptmrno , T_RPTMR *pk_rptmr);

Parameter

UINT	ptmrno	Physical Timer Number	Physical timer number
T_RPTMR*	pk_rptmr	Packet to Return Physical Timer Configuration Information	Pointer to the area to return the configuration information of the physical timer

Return Parameter

ER	ercd	Error Code	Error code

pk_rptmr Detail

UW	ptmrclk	Physical Timer Clock Frequency	Physical timer clock frequency
UW	maxcount	Maximum Count	Maximum count value
BOOL	defhdr	Handler Support	Whether physical timer handler is supported or not

Error Code

E_OK	Normal completion
E_PAR	Parameter error (ptmrno or pk_rptmr is invalid or cannot be used)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Gets the configuration information of the physical timer specified by ptmrno.

The retrievable configuration information includes the physical timer clock frequency ptmrclk, the maximum count value maxcount, and whether the support for physical timer handler exists defhdr.

ptmrclk indicates the clock frequency used to count up the target physical timer. If ptmrclk is set to 1, the clock is 1 Hz, and if it is set to MATH: 2⊠32 - 1, then the clock is MATH: 2⊠32 - 1 Hz (approximately 4 GHz). If the clock is long (less than 1 Hz), then ptmrclk is 0. If ptmrclk is other than 0, the physical timer count value is monotonically incremented by 1, from 0 to the upper limit value limit, at a constant time interval that is the inverse of ptmrclk.

maxcount is the maximum value that can be counted by the target physical timer, and also the maximum value that can be set as the upper limit value. Generally, maxcount is MATH: 2⊠16 - 1 for a 16-bit timer counter, and

MATH: 2⊠32 - 1 for a 32-bit timer counter, but it may be other value depending on the hardware or system configuration.

If `defhdr` is `TRUE`, the physical timer handler, which is started when the target physical timer count reaches the upper limit value, can be defined. If `defhdr` is `FALSE`, the physical timer handler for this physical timer cannot be defined.

If the physical timer specified by `ptmrno` does not exist or cannot be used, the E_PAR error occurs. For the physical timer number, a positive integer value is assigned in ascending order, so if the system has N physical timers, the E_PAR error occurs when `ptmrno` is 0 or larger than N.

Additional Notes

As the name of this function including "configuration" implies, the information `ptmrclk`, `maxcount`, and `defhdr` retrieved by this function are fixed statically by the hardware specification or the configuration at system start up, and it is assumed that they are not changed during the system operation. However, there is the possibility that the function to actively set or change the physical timer configuration (such as the clock frequency) is implemented in the future release or as additional implementation-dependent function. In such a case, the information retrieved by this function may become dynamic information that changes during the system operation. Such differences in usage depend heavily on the operation and usage, so it is better to absorb it in the upper library that uses the physical timer, rather than defining it as the specification of T-Kernel. For this reason, the T-Kernel specification does not specify the possibility that the configuration information retrieved by this function is changed during the system operation. That is, whether the information retrieved by this function may change during the operation is implementation-dependent.

Difference from T-Kernel 1.0

This API was added in T-Kernel 2.0.

5.10 Utility Functions

Utility functions are used commonly from general programs such as applications, middleware, and device drivers on the T-Kernel.

Utility functions are provided as library functions or C language macros.

Difference from T-Kernel 1.0
These functions were added in T-Kernel 2.0.

5.10.1 Set Object Name

API for setting object name is provided as C language macros. It can be called from a task-independent portion and while task dispatching and interrupts are disabled.

5.10.1.1 SetOBJNAME - Set Object Name

C Language Interface

```
#include <tk/tkernel.h>
```

void SetOBJNAME (void *exinf , CONST UB *name);

Parameter

void*	exinf	Extended Information	Variable to set as extended information
CONST UB*	name	Object Name	Object name to be set

Return Parameter

None

Error Code

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Interprets the ASCII string of four or less characters specified in name as a single 32-bit data to store it in exinf.

This API is defined as a C language macro and exinf is not a pointer. Write a variable directly.

Additional Notes

This API can be used to set a name (task name, etc.) for an individual object in T-Kernel as an ASCII string in the extended information exinf. When displaying the state of an object in the debugger, the object name set by this API can be shown by displaying the value in exinf as an ASCII string.

Example 5.7 Sample Usage of SetOBJNAME

```
T_CTSK  ctsk;
...
/* Set the object name "TEST" for the task ctsk */
SetOBJNAME(ctsk.exinf, "TEST");
task_id = tk_cre_tsk ( &ctsk );
```

Difference from T-Kernel 1.0

This API was added in T-Kernel 2.0.

5.10.2 Fast Lock and Multi-lock Libraries

Fast lock and multi-lock libraries are for performing exclusion control faster between multiple tasks in the device drivers or subsystems. In order to perform the exclusion control, while semaphore or mutex can be used, fast lock is implemented as the T-Kernel/SM library functions that processes the lock acquisition operation with specially higher speed when the task is not queued.

Among the fast lock and multi-lock libraries, the fast lock is a binary semaphore for mutual exclusion control faster than semaphores or mutexes. Fast multi-lock is one object built by combining 32 independent binary semaphores for mutual exclusion control each of which is distinguished by a lock number from 0 to 31.

For example, when exclusion control is performed at ten locations, one fast multi-lock can be created and then the binary semaphores with lock numbers from 0 to 9 can be used to perform exclusion control while ten fast locks can be used. While using ten fast locks bring faster result, the total required resources is lower when the fast multi-lock is used.

Additional Notes
Fast lock function is implemented by using counters that show the lock states and a semaphore. Fast multi-lock function is implemented by using a counter that shows the lock states and event flags. When the invoking task is not queued at the lock acquisition, it performs faster than the usual semaphores or event flags because only counter operation is performed. On the other hand, when the invoking task is queued at lock acquisition, it is not necessarily faster than the usual semaphores or event flags because it uses usual semaphores and event flags to manage transitions to waiting state or queues. Fast lock and multi-lock are effective when possibility of being queued is low due to mutual exclusion control.

Difference from T-Kernel 1.0
These libraries were added in T-Kernel 2.0.

5.10.2.1 CreateLock - Create Fast Lock

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = CreateLock (FastLock *lock , CONST UB *name);

Parameter

FastLock*	lock	Control Block of FastLock	Control block of fast lock
CONST UB*	name	Name of FastLock	Name of fast lock

Return Parameter

ER	ercd	Error Code	Error code

Error Codes

E_OK	Normal completion
E_NOMEM	Insufficient memory (memory for control block cannot be allocated)
E_LIMIT	Number of fast locks exceeds the system limit

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Creates a fast lock.

lock is a structure to control a fast lock. name is the name of the fast lock and can be NULL.

Fast lock is a binary semaphore used for mutual exclusion control and is implemented to be operated as fast as possible.

Difference from T-Kernel 1.0

This API was added in T-Kernel 2.0.

5.10.2.2 DeleteLock - Delete Fast Lock

C Language Interface

```
#include <tk/tkernel.h>
```

void DeleteLock (FastLock *lock);

Parameter

FastLock*	lock	Control Block of FastLock	Control block of fast lock

Return Parameter

None

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Deletes a fast lock.

Error detection is omitted for faster operation.

Difference from T-Kernel 1.0

This API was added in T-Kernel 2.0.

5.10.2.3 Lock - Lock Fast Lock

C Language Interface

```
#include <tk/tkernel.h>
```

void Lock (FastLock *lock);

Parameter

FastLock*	lock	Control Block of FastLock	Control block of fast lock

Return Parameter

None

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Locks a fast lock.

If the lock is already locked, the invoking task goes to the waiting state and is put in the task queue until it is unlocked. Tasks are queued in the priority order.

Error detection is omitted for faster operation.

Difference from T-Kernel 1.0

This API was added in T-Kernel 2.0.

5.10.2.4 Unlock - Unlock Fast Lock

C Language Interface

```
#include <tk/tkernel.h>
```

void Unlock (FastLock *lock);

Parameter

FastLock*	lock	Control Block of FastLock	Control block of fast lock

Return Parameter

None

Error Codes

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Unlocks a fast lock.
If there are tasks waiting for the fast lock, the first task in the task queue newly acquires the lock.
Error detection is omitted for faster operation.

Difference from T-Kernel 1.0

This API was added in T-Kernel 2.0.

5.10.2.5 CreateMLock - Create Fast Multi-lock

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = CreateMLock (FastMLock *lock , CONST UB *name);

Parameter

FastMLock*	lock	Control Block of FastMLock	Control block of fast multi-lock
CONST UB*	name	Name of FastMLock	Name of fast multi-lock

Return Parameter

ER	ercd	Error Code	Error code

Error Codes

E_OK	Normal completion
E_NOMEM	Insufficient memory (memory for control block cannot be allocated)
E_LIMIT	Number of fast multi-locks exceeds the system limit

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Creates a fast multi-lock.

lock is a structure to control a fast multi-lock. name is the name of the fast multi-lock and can be NULL.

Fast multi-lock is a list of 32 independent binary semaphores used for mutual exclusion control and is implemented to be operated as fast as possible. Each of the 32 binary semaphores is specified by a lock number from 0 to 31.

Difference from T-Kernel 1.0

This API was added in T-Kernel 2.0.

5.10.2.6 DeleteMLock - Delete Fast Multi-lock

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = DeleteMLock (FastMLock *lock);

Parameter

FastMLock*	lock	Control Block of FastMLock	Control block of fast multi-lock

Return Parameter

ER	ercd	Error Code	Error code

Error Codes

E_OK	Normal completion
E_PAR	Parameter error

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Deletes a fast multi-lock.

Difference from T-Kernel 1.0

This API was added in T-Kernel 2.0.

5.10.2.7 MLock - Lock Fast Multi-lock

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = MLock (FastMLock *lock , INT no);

Parameter

FastMLock*	lock	Control Block of FastMLock	Control block of fast multi-lock
INT	no	Lock Number	Lock number

Return Parameter

ER	ercd	Error Code	Error code

Error Codes

E_OK	Normal completion
E_PAR	Parameter error
E_DLT	Waiting object was deleted
E_RLWAI	Waiting state was forcibly released
E_CTX	Context error

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Locks a fast multi-lock.

no is a lock number from 0 to 31.

If the lock is already locked with the same lock number, the invoking task goes to the waiting state and is put in the task queue until it is unlocked with the same lock number. Tasks are queued in the priority order.

Difference from T-Kernel 1.0

This API was added in T-Kernel 2.0.

5.10.2.8 MLockTmo - Lock Fast Multi-lock (with Timeout)

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = MLockTmo (FastMLock *lock , INT no , TMO tmout);

Parameter

FastMLock*	lock	Control Block of FastMLock	Control block of fast multi-lock
INT	no	Lock Number	Lock number
TMO	tmout	Timeout	Timeout (ms)

Return Parameter

ER	ercd	Error Code	Error code

Error Codes

E_OK	Normal completion
E_PAR	Parameter error
E_DLT	Waiting object was deleted
E_RLWAI	Waiting state was forcibly released
E_TMOUT	Timeout
E_CTX	Context error

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Locks a fast multi-lock with timeout.

This API is identical to MLock(), except that it can specify the timeout interval in `tmout`. If the lock cannot be acquired before the timeout interval specified in `tmout` has elapsed, E_TMOUT is returned.

Difference from T-Kernel 1.0

This API was added in T-Kernel 2.0.

5.10.2.9 MLockTmo_u - Lock Fast Multi-lock (with Timeout, in Microseconds)

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = MLockTmo_u (FastMLock *lock , INT no , TMO_U tmout_u);

Parameter

FastMLock*	lock	Control Block of FastMLock	Control block of fast multi-lock
INT	no	Lock Number	Lock number
TMO_U	tmout_u	Timeout	Timeout (in microseconds)

Return Parameter

ER	ercd	Error Code	Error code

Error Codes

E_OK	Normal completion
E_PAR	Parameter error
E_DLT	Waiting object was deleted
E_RLWAI	Waiting state was forcibly released
E_TMOUT	Timeout
E_CTX	Context error

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Locks a fast multi-lock with timeout in microseconds.

This API is identical to MLockTmo(), except that the timeout interval is specified with a 64-bit value in microseconds.

Difference from T-Kernel 1.0

This API was added in T-Kernel 2.0.

5.10.2.10 MUnlock - Unlock Fast Multi-lock

C Language Interface

```
#include <tk/tkernel.h>
```

ER ercd = MUnlock (FastMLock *lock , INT no);

Parameter

FastMLock*	lock	Control Block of FastMLock	Control block of fast multi-lock
INT	no	Lock Number	Lock number

Return Parameter

ER	ercd	Error Code	Error code

Error Codes

E_OK	Normal completion

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	NO

Description

Unlocks a fast multi-lock.

no is a lock number from 0 to 31.

If there are tasks in the waiting state for the same lock number, the first task in the task queue newly acquires the lock.

Difference from T-Kernel 1.0

This API was added in T-Kernel 2.0.

5.11 Subsystem and Device Driver Starting

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 system start processing task or termination processing task (protection level 0). In some 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, entry routines are called one at a time for each 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 initializing 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 T-Kernel, specifying parameters, and specifying the order in which entry routines are called are all dependent on the T-Kernel implementation.

5.11.1 Startup Processing

Parameter

INT	ac	Number of parameters (≧ 0)
UB*	av	Parameters (string)

Return Parameter

Return Codes	Error Code

Description

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 T-Kernel implementation, the subsystem or device driver may be deleted from memory, so error must not

be returned while the subsystem or device driver is in registered state. The registration must first be erased before returning an 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 last element of the array (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 what name is stored is implementation-dependent. It is also possible to have no name (blank string "").

Parameters at and after av[1] are defined for each subsystem and device driver.

After exit from the entry routine, the character string space specified by av is deleted, so parameters must be saved to a different location if necessary.

5.11.2 Termination Processing

Parameter

INT	ac	-1
UB*	av	NULL

Return Parameter

Return Codes	Error Code

Description

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 as much as possible. If some of the processing could not be completed normally, 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, ordinarily behavior is not guaranteed in the case of requests issued during termination processing.

Chapter 6

T-Kernel/DS Functions

This chapter describes details of the functions provided by T-Kernel/DS (Debugger Support).

T-Kernel/DS provides functions enabling a debugger to reference T-Kernel internal states and run a trace. The functions provided by T-Kernel/DS are only for debugger use and not for use by applications or other programs.

Overall Note and Supplement

- Except where otherwise noted, T-Kernel/DS system 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 particular implementations.

- When T-Kernel/DS system calls (td_...) are invoked in interrupts disabled state, they are processed without enabling interrupts. Other kernel states likewise remain unchanged during this processing. Changes in kernel states may occur if a service call is invoked while interrupts or dispatching are enabled, since the kernel continues operating.

- T-Kernel/DS system 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).

- Error codes such as E_PAR, E_MACV, and E_CTX that can be returned in many situations are not described here always unless there is some special reason for doing so.

6.1 Kernel Internal State Acquisition Functions

Kernel internal state reference functions are functions for enabling 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.

6.1.1 td_lst_tsk - Reference Task ID List

C Language Interface

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_tsk (ID list[], INT nent);

Parameter

ID	list[]		List	Location of task ID list
INT	nent		Number of List Entries	Maximum number of entries in list

Return Parameter

INT	ct		Count	Number of used tasks
		or	Error Code	Error code

Error Code

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the list of the IDs of the currently used tasks, and puts in list up to nent IDs. The number of the used tasks is passed in the return code. If return code > nent, this means not all task IDs could be retrieved.

6.1.2 td_lst_sem - Reference Semaphore ID List

C Language Interface

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_sem (ID list[], INT nent);

Parameter

ID	list[]		List	Location of semaphore ID list
INT	nent		Number of List Entries	Maximum number of entries in list

Return Parameter

INT	ct		Count	Number of used semaphores
		or	Error Code	Error code

Error Code

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the list of the IDs of the currently used semaphores, and puts in list up to nent IDs. The number of the used semaphores is passed in the return code. If return code > nent, this means not all semaphore IDs could be retrieved.

6.1.3 td_lst_flg - Reference Event Flag ID List

C Language Interface

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_flg (ID list[], INT nent);

Parameter

ID	list[]		List	Location of event flag ID list
INT	nent		Number of List Entries	Maximum number of entries in list

Return Parameter

INT	ct		Count	Number of used event flags
		or	Error Code	Error code

Error Code

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the list of the IDs of the currently used event flags, and puts in list up to nent IDs. The number of the used event flags is passed in the return code. If return code > nent, this means not all event flag IDs could be retrieved.

6.1.4 td_lst_mbx - Reference Mailbox ID List

C Language Interface

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_mbx (ID list[], INT nent);

Parameter

ID	list[]		List	Location of mailbox ID list
INT	nent		Number of List Entries	Maximum number of entries in list

Return Parameter

INT	ct		Count	Number of used mailboxes
		or	Error Code	Error code

Error Code

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the list of the IDs of the currently used mailboxes, and puts in list up to nent IDs. The number of the used mailboxes is passed in the return code. If return code > nent, this means not all mailbox IDs could be retrieved.

6.1.5 td_lst_mtx - Reference Mutex ID List

C Language Interface

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_mtx (ID list[], INT nent);

Parameter

ID	list[]		List	Location of mutex ID list
INT	nent		Number of List Entries	Maximum number of entries in list

Return Parameter

INT	ct		Count	Number of used mutexes
		or	Error Code	Error code

Error Code

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the list of the IDs of the currently used mutexes, and puts in list up to nent IDs. The number of the used mutexes is passed in the return code. If return code > nent, this means not all mutex IDs could be retrieved.

6.1.6 td_lst_mbf - Reference Message Buffer ID List

C Language Interface

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_mbf (ID list[], INT nent);

Parameter

ID	list[]		List	Location of message buffer ID list
INT	nent		Number of List Entries	Maximum number of entries in list

Return Parameter

INT	ct		Count	Number of used message buffers
		or	Error Code	Error code

Error Code

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the list of the IDs of the currently used message buffers, and puts in list up to nent IDs. The number of the used message buffers is passed in the return code. If return code > nent, this means not all message buffer IDs could be retrieved.

6.1.7 td_lst_por - Reference Rendezvous Port ID List

C Language Interface

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_por (ID list[], INT nent);

Parameter

ID	list[]		List	Location of rendezvous port ID list
INT	nent		Number of List Entries	Maximum number of entries in list

Return Parameter

INT	ct		Count	Number of used rendezvous ports
		or	Error Code	Error code

Error Code

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the list of the IDs of the currently used rendezvous ports, and puts in list up to nent IDs. The number of the used rendezvous ports is passed in the return code. If return code > nent, this means not all rendezvous port IDs could be retrieved.

6.1.8 td_lst_mpf - Reference Fixed-size Memory Pool ID List

C Language Interface

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_mpf (ID list[], INT nent);

Parameter

ID	list[]		List	Location of fixed-size memory pool ID list
INT	nent		Number of List Entries	Maximum number of entries in list

Return Parameter

INT	ct		Count	Number of used fixed-size memory pools
		or	Error Code	Error code

Error Code

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the list of the IDs of the currently used fixed-size memory pools, and puts in list up to nent IDs. The number of the used fixed-size memory pools is passed in the return code. If return code > nent, this means not all fixed-size memory pool IDs could be retrieved.

6.1.9 td_lst_mpl - Reference Variable-size Memory Pool ID List

C Language Interface

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_mpl (ID list[], INT nent);

Parameter

ID	list[]		List	Location of variable-size memory pool ID list
INT	nent		Number of List Entries	Maximum number of entries in list

Return Parameter

INT	ct		Count	Number of used variable-size memory pools
		or	Error Code	Error code

Error Code

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the list of the IDs of the currently used variable-size memory pools, and puts in list up to nent IDs. The number of the used variable-size memory pools is passed in the return code. If return code > nent, this means not all variable-size memory pool IDs could be retrieved.

6.1.10 td_lst_cyc - Reference Cyclic Handler ID List

C Language Interface

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_cyc (ID list[], INT nent);

Parameter

ID	list[]		List	Location of cyclic handler ID list
INT	nent		Number of List Entries	Maximum number of entries in list

Return Parameter

INT	ct		Count	Number of used cyclic handlers
		or	Error Code	Error code

Error Code

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the list of the IDs of the currently used cyclic handlers, and puts in list up to nent IDs. The number of the used cyclic handlers is passed in the return code. If return code > nent, this means not all cyclic handler IDs could be retrieved.

6.1.11 td_lst_alm - Reference Alarm Handler ID List

C Language Interface

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_alm (ID list[], INT nent);

Parameter

ID	list[]		List	Location of alarm handler ID list
INT	nent		Number of List Entries	Maximum number of entries in list

Return Parameter

INT	ct		Count	Number of used alarm handlers
		or	Error Code	Error code

Error Code

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the list of the IDs of the currently used alarm handlers, and puts in list up to nent IDs. The number of the used alarm handlers is passed in the return code. If return code > nent, this means not all alarm handler IDs could be retrieved.

6.1.12 td_lst_ssy - Reference Subsystem ID List

C Language Interface

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_ssy (ID list[], INT nent);

Parameter

ID	list[]		List	Location of subsystem ID list
INT	nent		Number of List Entries	Maximum number of entries in list

Return Parameter

INT	ct		Count	Number of used subsystems
		or	Error Code	Error code

Error Code

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the list of the IDs of the currently used subsystems, and puts in list up to nent IDs. The number of the used subsystems is passed in the return code. If return code > nent, this means not all subsystem IDs could be retrieved.

6.1.13 td_rdy_que - Reference Task Precedence

C Language Interface

```
#include <tk/dbgspt.h>
```

INT ct = td_rdy_que (PRI pri , ID list[], INT nent);

Parameter

PRI	pri	Task Priority	Task priority
ID	list[]	Task ID List	Location of task ID list
INT	nent	Number of List Entries	Maximum number of entries in list

Return Parameter

INT	ct	Count	Number of tasks with priority pri in a run state
	or	Error Code	Error code

Error Code

E_PAR	Parameter error (pri is invalid or cannot be used)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets a list of IDs of the tasks in a run state (READY state or RUNNING state) whose task priority is pri, arranged in the order from the highest to the lowest precedence.

This function stores in list up to nent task IDs, arranged in the 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 not all task IDs could be retrieved.

6.1.14 td_sem_que - Reference Semaphore Queue

C Language Interface

```
#include <tk/dbgspt.h>
```

INT ct = td_sem_que (ID semid , ID list[], INT nent);

Parameter

ID	semid	Semaphore ID	Target semaphore ID
ID	list[]	Task ID List	Location of waiting task IDs
INT	nent	Number of List Entries	Maximum number of entries in list

Return Parameter

INT	ct		Count	Number of waiting tasks
		or	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the list of the IDs of the queued tasks waiting for a semaphore specified in semid. This function stores in list up to nent task IDs, arranged in the order in which tasks are queued, starting from the first task in the semaphore queue. The number of the tasks in the semaphore queue is passed in the return code. If return code > nent, this means not all task IDs could be retrieved.

6.1.15 td_flg_que - Reference Event Flag Queue

C Language Interface

```
#include <tk/dbgspt.h>
```

INT ct = td_flg_que (ID flgid , ID list[], INT nent);

Parameter

ID	flgid	EventFlag ID	Target event flag ID
ID	list[]	Task ID List	Location of waiting task IDs
INT	nent	Number of List Entries	Maximum number of entries in list

Return Parameter

INT	ct		Count	Number of waiting tasks
		or	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the list of the IDs of the queued tasks waiting for an event flag specified in flgid. This function stores in list up to nent task IDs, arranged in the order in which tasks are queued, starting from the first task in the event flag queue. The number of the tasks in the event flag queue is passed in the return code. If return code > nent, this means not all task IDs could be retrieved.

6.1.16 td_mbx_que - Reference Mailbox Queue

C Language Interface

```
#include <tk/dbgspt.h>
```

INT ct = td_mbx_que (ID mbxid , ID list[], INT nent);

Parameter

ID	mbxid	Mailbox ID	Target mailbox ID
ID	list[]	Task ID List	Location of waiting task IDs
INT	nent	Number of List Entries	Maximum number of entries in list

Return Parameter

INT	ct		Count	Number of waiting tasks
		or	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the list of the IDs of the queued tasks waiting for a mailbox specified in mbxid. This function stores in list up to nent task IDs, arranged in the order in which tasks are queued, starting from the first task in the mailbox queue. The number of the tasks in the mailbox queue is passed in the return code. If return code > nent, this means not all task IDs could be retrieved.

6.1.17 td_mtx_que - Reference Mutex Queue

C Language Interface

```
#include <tk/dbgspt.h>
```

INT ct = td_mtx_que (ID mtxid , ID list[], INT nent);

Parameter

ID	mtxid	Mutex ID	Target mutex ID
ID	list[]	Task ID List	Location of waiting task IDs
INT	nent	Number of List Entries	Maximum number of entries in list

Return Parameter

INT	ct		Count	Number of waiting tasks
		or	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the list of the IDs of the queued tasks waiting for a mutex specified in mtxid. This function stores in list up to nent task IDs, arranged in the order in which tasks are queued, starting from the first task in the mutex queue. The number of the tasks in the mutex queue is passed in the return code. If return code > nent, this means not all task IDs could be retrieved.

6.1.18 td_smbf_que - Reference Message Buffer Send Queue

C Language Interface

```
#include <tk/dbgspt.h>
```

INT ct = td_smbf_que (ID mbfid , ID list[], INT nent);

Parameter

ID	mbfid	Message Buffer ID	Target message buffer ID
ID	list[]	Task ID List	Location of waiting task IDs
INT	nent	Number of List Entries	Maximum number of entries in list

Return Parameter

INT	ct		Count	Number of waiting tasks
		or	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the list of the IDs of the queued tasks waiting for sending a message to a message buffer specified in mbfid. This function stores in list up to nent task IDs, arranged in the order in which tasks are queued, starting from the first task in the message buffer send queue. The number of the tasks in the message buffer send queue is passed in the return code. If return code > nent, this means not all task IDs could be retrieved.

6.1.19 td_rmbf_que - Reference Message Buffer Receive Queue

C Language Interface

```
#include <tk/dbgspt.h>
```

INT ct = td_rmbf_que (ID mbfid , ID list[], INT nent);

Parameter

ID	mbfid	Message Buffer ID	Target message buffer ID
ID	list[]	Task ID List	Location of waiting task IDs
INT	nent	Number of List Entries	Maximum number of entries in list

Return Parameter

INT	ct		Count	Number of waiting tasks
		or	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the list of the IDs of the queued tasks waiting for receiving a message from a message buffer specified in mbfid. This function stores in list up to nent task IDs, arranged in the order in which tasks are queued, starting from the first task in the message buffer receive queue. The number of the tasks in the message buffer receive queue is passed in the return code. If return code > nent, this means not all task IDs could be retrieved.

6.1.20 td_cal_que - Reference Call Queue

C Language Interface

```
#include <tk/dbgspt.h>
```

INT ct = td_cal_que (ID porid , ID list[], INT nent);

Parameter

ID	porid	Port ID	Target rendezvous port ID
ID	list[]	Task ID List	Location of waiting task IDs
INT	nent	Number of List Entries	Maximum number of entries in list

Return Parameter

INT	ct		Count	Number of waiting tasks
		or	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the list of the IDs of the queued tasks waiting for rendezvous call at a port specified in porid. This function stores in list up to nent task IDs, arranged in the order in which tasks are queued, starting from the first task in the rendezvous call queue. The number of the tasks in the rendezvous call queue is passed in the return code. If return code > nent, this means not all task IDs could be retrieved.

6.1.21 td_acp_que - Reference Accept Queue

C Language Interface

```
#include <tk/dbgspt.h>
```

INT ct = td_acp_que (ID porid , ID list[], INT nent);

Parameter

ID	porid	Port ID	Target rendezvous port ID
ID	list[]	Task ID List	Location of waiting task IDs
INT	nent	Number of List Entries	Maximum number of entries in list

Return Parameter

INT	ct		Count	Number of waiting tasks
		or	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the list of the IDs of the queued tasks waiting for rendezvous acceptance at a port specified in porid. This function stores in list up to nent task IDs, arranged in the order in which tasks are queued, starting from the first task in the rendezvous acceptance queue. The number of the tasks in the rendezvous acceptance queue is passed in the return code. If return code > nent, this means not all task IDs could be retrieved.

6.1.22 td_mpf_que - Reference Fixed-size Memory Pool Queue

C Language Interface

```
#include <tk/dbgspt.h>
```

INT ct = td_mpf_que (ID mpfid , ID list[], INT nent);

Parameter

ID	mpfid	Memory Pool ID	Target fixed-size memory pool ID
ID	list[]	Task ID List	Location of waiting task IDs
INT	nent	Number of List Entries	Maximum number of entries in list

Return Parameter

INT	ct		Count	Number of waiting tasks
		or	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the list of the IDs of the queued tasks waiting for allocation in a fixed-size memory pool specified in mpfid. This function stores in list up to nent task IDs, arranged in the order in which tasks are queued, starting from the first task in the fixed-size memory pool queue. The number of the tasks in the fixed-size memory pool queue is passed in the return code. If return code > nent, this means not all task IDs could be retrieved.

6.1.23 td_mpl_que - Reference Variable-size Memory Pool Queue

C Language Interface

```
#include <tk/dbgspt.h>
```

INT ct = td_mpl_que (ID mplid , ID list[], INT nent);

Parameter

ID	mplid	Memory Pool ID	Target variable-size memory pool ID
ID	list[]	Task ID List	Location of waiting task IDs
INT	nent	Number of List Entries	Maximum number of entries in list

Return Parameter

INT	ct		Count	Number of waiting tasks
		or	Error Code	Error code

Error Code

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)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the list of the IDs of the queued tasks waiting for allocation in a variable-size memory pool specified in mplid. This function stores in list up to nent task IDs, arranged in the order in which tasks are queued, starting from the first task in the variable-size memory pool queue. The number of the tasks in the variable-size memory pool queue is passed in the return code. If return code > nent, this means not all task IDs could be retrieved.

6.1.24 td_ref_tsk - Reference Task Status

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_tsk (ID tskid , TD_RTSK *rtsk);

Parameter

ID	tskid	Task ID	Target task ID (TSK_SELF can be specified)
TD_RTSK*	rtsk	Packet to Return Task Status	Pointer to the area to return the task status

Return Parameter

ER	ercd	Error Code	Error code

rtsk Detail:

void*	exinf	Extended Information	Extended information
PRI	tskpri	Task Priority	Current priority
PRI	tskbpri	Task Base Priority	Base priority
UINT	tskstat	Task State	Task States
UINT	tskwait	Task Wait Factor	Wait factor
ID	wid	Waiting Object ID	Waiting object ID
INT	wupcnt	Wakeup Count	Wakeup request queuing count
INT	suscnt	Suspend Count	Suspend request nesting count
RELTIM	slicetime	Slice Time	Maximum continuous run time (in ms)
UINT	waitmask	Wait Mask	Disabled wait factors
UINT	texmask	Task Exception Mask	Allowed task exceptions
UINT	tskevent	Task Event	Raised task event
FP	task	Task Start Address	Task start address
INT	stksz	User Stack Size	User stack size (in bytes)
INT	sstksz	System Stack Size	System stack size (in bytes)
void*	istack	Initial User Stack Pointer	User stack pointer initial value
void*	isstack	Initial System Stack Pointer	System stack pointer initial value

Error Code

E_OK	Normal completion
E_ID	Bad identifier
E_NOEXS	Object does not exist

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

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.

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 stack pointer current position.

`slicetime` in the task status information (TD_RTSK) returns a value rounded to milliseconds. To know the value in microseconds, call td_ref_tsk_u.

6.1.25 td_ref_tsk_u - Reference Task Status (Microseconds)

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_tsk_u (ID tskid , TD_RTSK_U *rtsk_u);

Parameter

ID	tskid	Task ID	Target task ID (TSK_SELF can be specified)
TD_RTSK_U*	rtsk_u	Packet to Return Task Status	Pointer to the area to return the task status

Return Parameter

ER	ercd	Error Code	Error code

rtsk_u Detail:

void*	exinf	Extended Information	Extended information
PRI	tskpri	Task Priority	Current priority
PRI	tskbpri	Task Base Priority	Base priority
UINT	tskstat	Task State	Task States
UINT	tskwait	Task Wait Factor	Wait factor
ID	wid	Waiting Object ID	Waiting object ID
INT	wupcnt	Wakeup Count	Wakeup request queuing count
INT	suscnt	Suspend Count	Suspend request nesting count
RELTIM_U	slicetime_u	Slice Time	Maximum continuous run time (in microseconds)
UINT	waitmask	Wait Mask	Disabled wait factors
UINT	texmask	Task Exception Mask	Allowed task exceptions
UINT	tskevent	Task Event	Raised task event
FP	task	Task Start Address	Task start address
INT	stksz	User Stack Size	User stack size (in bytes)
INT	sstksz	System Stack Size	System stack size (in bytes)
void*	istack	Initial User Stack Pointer	User stack pointer initial value
void*	isstack	Initial System Stack Pointer	System stack pointer initial value

Error Code

E_OK	Normal completion
E_ID	Bad identifier
E_NOEXS	Object does not exist

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

This system call takes `slicetime_u` in 64-bit microseconds instead of the return parameter `slicetime` of td_ref_tsk.

The specification of this system call is same as that of td_ref_tsk, except that a field in the return parameter is replaced with `slicetime_u`. For more details, see the description of td_ref_tsk.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

6.1.26 td_ref_tex - Reference Task Exception Status

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_tex (ID tskid , TD_RTEX *pk_rtex);

Parameter

ID	tskid	Task ID	Target task ID (TSK_SELF can be specified)
TD_RTEX*	pk_rtex	Packet to Return Task Exception Status	Pointer to the area to return the task exception status

Return Parameter

ER	ercd	Error Code	Error code

pk_rtex Detail:

UINT	pendtex	Pending Task Exception	Pending task exceptions
UINT	texmask	Task Exception Mask	Allowed task exceptions

Error Code

E_OK	Normal completion
E_ID	Bad identifier
E_NOEXS	Object does not exist

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the task exception status. This is similar to tk_ref_tex().

6.1.27 td_ref_sem - Reference Semaphore Status

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_sem (ID semid , TD_RSEM *rsem);

Parameter

ID	semid	Semaphore ID	Target semaphore ID
TD_RSEM*	rsem	Packet to Return Semaphore Status	Pointer to the area to return the semaphore status

Return Parameter

ER	ercd	Error Code	Error code

rsem Detail:

void*	exinf	Extended Information	Extended information
ID	wtsk	Waiting Task ID	Waiting task ID
INT	semcnt	Semaphore Count	current semaphore count value

Error Code

E_OK	Normal completion
E_ID	Bad identifier
E_NOEXS	Object does not exist

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

References the semaphore status. This is similar to tk_ref_sem().

6.1.28 td_ref_flg - Reference Event Flag Status

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_flg (ID flgid , TD_RFLG *rflg);

Parameter

ID	flgid	EventFlag ID	Target event flag ID
TD_RFLG*	rflg	Packet to Return EventFlag Status	Pointer to the area to return the event flag status

Return Parameter

ER	ercd	Error Code	Error code

`rflg` Detail:

void*	exinf	Extended Information	Extended information
ID	wtsk	Waiting Task ID	Waiting task ID
UINT	flgptn	EventFlag Bit Pattern	The current event flag bit pattern

Error Code

E_OK	Normal completion
E_ID	Bad identifier
E_NOEXS	Object does not exist

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

References the event flag status. This is similar to tk_ref_flg().

6.1.29 td_ref_mbx - Reference Mailbox Status

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_mbx (ID mbxid , TD_RMBX *rmbx);

Parameter

ID	mbxid	Mailbox ID	Target mailbox ID
TD_RMBX*	rmbx	Packet to Return Mailbox Status	Pointer to the area to return the mailbox status

Return Parameter

ER	ercd	Error Code	Error code

rmbx Detail:

void*	exinf	Extended Information	Extended information
ID	wtsk	Waiting Task ID	Waiting task ID
T_MSG*	pk_msg	Packet of Message	Next message to be received

Error Code

E_OK	Normal completion
E_ID	Bad identifier
E_NOEXS	Object does not exist

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

References the mailbox status. This is similar to tk_ref_mbx().

6.1.30 td_ref_mtx - Refer Mutex Status

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_mtx (ID mtxid , TD_RMTX *rmtx);

Parameter

ID	mtxid	Mutex ID	Target mutex ID
TD_RMTX*	rmtx	Packet to Return Mutex Status	Pointer to the area to return the mutex status

Return Parameter

ER	ercd	Error Code	Error code

rmtx Detail:

void*	exinf	Extended Information	Extended information
ID	htsk	Locking Task ID	ID of task locking the mutex
ID	wtsk	Lock Waiting Task ID	ID of tasks waiting to lock the mutex

Error Code

E_OK	Normal completion
E_ID	Bad identifier
E_NOEXS	Object does not exist

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

References the mutex status. This is similar to tk_ref_mtx().

6.1.31 td_ref_mbf - Reference Message Buffer Status

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_mbf (ID mbfid , TD_RMBF *rmbf);

Parameter

ID	mbfid	Message Buffer ID	Target message buffer ID
TD_RMBF*	rmbf	Packet to Return Message Buffer Status	Pointer to the area to return the message buffer status

Return Parameter

ER	ercd	Error Code	Error code

rmbf Detail:

void*	exinf	Extended Information	Extended information
ID	wtsk	Waiting Task ID	Receive waiting task ID
ID	stsk	Send Waiting Task ID	Send waiting task ID
INT	msgsz	Message Size	Size of the next message to be received (in bytes)
INT	frbufsz	Free Buffer Size	Free buffer size (in bytes)
INT	maxmsz	Maximum Message Size	Maximum message size (in bytes)

Error Code

E_OK	Normal completion
E_ID	Bad identifier
E_NOEXS	Object does not exist

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

References the message buffer status. This is similar to tk_ref_mbf().

6.1.32 td_ref_por - Reference Port Status

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_por (ID porid , TD_RPOR *rpor);

Parameter

ID	porid	Port ID	Target rendezvous port ID
TD_RPOR*	rpor	Packet to Return Port Status	Pointer to the area to return the rendezvous port status

Return Parameter

ER	ercd	Error Code	Error code

rpor Detail:

void*	exinf	Extended Information	Extended information
ID	wtsk	Waiting Task ID	Call waiting task ID
ID	atsk	Accept Waiting Task ID	Accept waiting task ID
INT	maxcmsz	Maximum Call Message Size	Maximum call message size (in bytes)
INT	maxrmsz	Maximum Reply Message Size	Maximum reply message size (in bytes)

Error Code

E_OK	Normal completion
E_ID	Bad identifier
E_NOEXS	Object does not exist

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

References the rendezvous port status. This is similar to tk_ref_por().

6.1.33 td_ref_mpf - Reference Fixed-size Memory Pool Status

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_mpf (ID mpfid , TD_RMPF *rmpf);

Parameter

ID	mpfid	Memory Pool ID	Target fixed-size memory pool ID
TD_RMPF*	rmpf	Packet to Return Memory Pool Status	Pointer to the area to return the memory pool status

Return Parameter

ER	ercd	Error Code	Error code

rmpf Detail:

void*	exinf	Extended Information	Extended information
ID	wtsk	Waiting Task ID	Waiting task ID
INT	frbcnt	Free Block Count	Free block count

Error Code

E_OK	Normal completion
E_ID	Bad identifier
E_NOEXS	Object does not exist

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

References the fixed-size memory pool status. This is similar to tk_ref_mpf().

6.1.34 td_ref_mpl - Reference Variable-size Memory Pool Status

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_mpl (ID mplid , TD_RMPL *rmpl);

Parameter

ID	mplid	Memory Pool ID	Target variable-size memory pool ID
TD_RMPL*	rmpl	Packet to Return Memory Pool Status	Pointer to the area to return the memory pool status

Return Parameter

ER	ercd	Error Code	Error code

rmpl Detail:

void*	exinf	Extended Information	Extended information
ID	wtsk	Waiting Task ID	Waiting task ID
INT	frsz	Free Memory Size	Free memory size (in bytes)
INT	maxsz	Max Memory Size	Maximum memory space size (in bytes)

Error Code

E_OK	Normal completion
E_ID	Bad identifier
E_NOEXS	Object does not exist

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

References the variable-size memory pool status. This is similar to tk_ref_mpl().

6.1.35 td_ref_cyc - Reference Cyclic Handler Status

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_cyc (ID cycid , TD_RCYC *rcyc);

Parameter

ID	cycid	Cyclic Handler ID	Target cyclic handler ID
TD_RCYC*	rcyc	Packet to Return Cyclic Handler Status	Pointer to the area to return the cyclic handler status

Return Parameter

ER	ercd	Error Code	Error code

rcyc Detail:

void*	exinf	Extended Information	Extended information
RELTIM	lfttim	Left Time	Time remaining until the next handler starts (ms)
UINT	cycstat	Cyclic Handler Status	Cyclic handler activation state

Error Code

E_OK	Normal completion
E_ID	Bad identifier
E_NOEXS	Object does not exist

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

References the cyclic handler status. This is similar to tk_ref_cyc().

The time remaining lfttim returned in the cyclic handler status information (TD_RCYC) obtained by td_ref_cyc is a value rounded to milliseconds. To know the value in microseconds, call td_ref_cyc_u.

6.1.36 td_ref_cyc_u - Reference Cyclic Handler Status (Microseconds)

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_cyc_u (ID cycid , TD_RCYC_U *rcyc_u);

Parameter

ID	cycid	Cyclic Handler ID	Target cyclic handler ID
TD_RCYC_U*	rcyc_u	Packet to Return Cyclic Handler Status	Pointer to the area to return the cyclic handler status

Return Parameter

ER	ercd	Error Code	Error code

rcyc_u Detail:

void*	exinf	Extended Information	Extended information
RELTIM_U	lfttim_u	Left Time	Time remaining until the next handler starts (microseconds)
UINT	cycstat	Cyclic Handler Status	Cyclic handler activation state

Error Code

E_OK	Normal completion
E_ID	Bad identifier
E_NOEXS	Object does not exist

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

This system call takes 64-bit lfttim_u in microseconds instead of the return parameter lfttim of td_ref_cyc.

The specification of this system call is same as that of td_ref_cyc, except that the return parameter is replaced with lfttim_u. For more details, see the description of td_ref_cyc.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

6.1.37 td_ref_alm - Reference Alarm Handler Status

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_alm (ID almid , TD_RALM *ralm);

Parameter

ID	almid	Alarm Handler ID	Target alarm handler ID
TD_RALM*	ralm	Packet to Return Alarm Handler Status	Pointer to the area to return the alarm handler status

Return Parameter

ER	ercd	Error Code	Error code

ralm Detail:

void*	exinf	Extended Information	Extended information
RELTIM	lfttim	Left Time	Time remaining until the handler starts (ms)
UINT	almstat	Alarm Handler Status	Alarm handler activation state

Error Code

E_OK	Normal completion
E_ID	Bad identifier
E_NOEXS	Object does not exist

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

References the alarm handler status. This is similar to tk_ref_alm().

The time remaining lfttim returned in the alarm handler status information (TD_RALM) obtained by td_ref_alm is a value rounded to milliseconds. To know the value in microseconds, call td_ref_alm_u.

6.1.38 td_ref_alm_u - Reference Alarm Handler Status (Microseconds)

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_alm_u (ID almid , TD_RALM_U *ralm_u);

Parameter

ID	almid	Alarm Handler ID	Target alarm handler ID
TD_RALM_U*	ralm_u	Packet to Return Alarm Handler Status	Pointer to the area to return the alarm handler status

Return Parameter

ER	ercd	Error Code	Error code

ralm_u Detail:

void*	exinf	Extended Information	Extended information
RELTIM_U	lfttim_u	Left Time	Time remaining until the handler starts (microseconds)
UINT	almstat	Alarm Handler Status	Alarm handler activation state

Error Code

E_OK	Normal completion
E_ID	Bad identifier
E_NOEXS	Object does not exist

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

This system call takes 64-bit lfttim_u in microseconds instead of the return parameter lfttim of td_ref_alm.

The specification of this system call is same as that of td_ref_alm, except that the return parameter is replaced with lfttim_u. For more details, see the description of td_ref_alm.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

6.1.39 td_ref_sys - Reference System Status

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_sys (TD_RSYS *pk_rsys);

Parameter

TD_RSYS*	pk_rsys	Packet to Return System Status	Pointer to the area to return the system status

Return Parameter

ER	ercd	Error Code	Error code

pk_rsys Detail:

INT	sysstat	System State	System State
ID	runtskid	Running Task ID	ID of the task currently in RUNNING state
ID	schedtskid	Scheduled Task ID	ID of the task scheduled to run next

Error Code

E_OK	Normal completion

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets the system status. This is similar to tk_ref_sys().

6.1.40 td_ref_ssy - Reference Subsystem Status

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_ssy (ID ssid , TD_RSSY *rssy);

Parameter

ID	ssid	Subsystem ID	Target subsystem ID
TD_RSSY*	rssy	Packet to Return Subsystem Status	Pointer to the area to return the subsystem definition information

Return Parameter

ER	ercd	Error Code	Error code

rssy Detail:

PRI	ssypri	Subsystem Priority	Subsystem priority
INT	resblksz	Resource Control Block Size	Resource control block size (in bytes)

Error Code

E_OK	Normal completion
E_ID	Bad identifier
E_NOEXS	Object does not exist

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

References the subsystem status. This is similar to tk_ref_ssy().

6.1.41 td_inf_tsk - Reference Task Statistics

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_inf_tsk (ID tskid , TD_ITSK *pk_itsk , BOOL clr);

Parameter

ID	tskid	Task ID	Target task ID (TSK_SELF can be specified)
TD_ITSK*	pk_itsk	Packet to Return Task Statistics	Pointer to the area to return the task statistics
BOOL	clr	Clear	Task statistics clear flag

Return Parameter

ER	ercd	Error Code	Error code

pk_itsk Detail:

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

Error Code

E_OK	Normal completion
E_ID	Bad identifier
E_NOEXS	Object does not exist

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Gets task statistics. This is similar to tk_inf_tsk(). If clr = TRUE (≠ 0), the cumulative information is reset (cleared to 0) after the information is obtained.

stime and utime in the task statistics (TD_ITSK) return values rounded to milliseconds. To know the value in microseconds, call td_inf_tsk_u.

6.1.42 td_inf_tsk_u - Reference Task Statistics (Microseconds)

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_inf_tsk_u (ID tskid , TD_ITSK_U *itsk_u , BOOL clr);

Parameter

ID	tskid	Task ID	Target task ID (TSK_SELF can be specified)
TD_ITSK_U*	itsk_u	Packet to Return Task Statistics	Pointer to the area to return the task statistics
BOOL	clr	Clear	Task statistics clear flag

Return Parameter

ER	ercd	Error Code	Error code

itsk_u Detail:

RELTIM_U	stime_u	System Time	Cumulative system-level run time (in microseconds)
RELTIM_U	utime_u	User Time	Cumulative user-level run time (in microseconds)

Error Code

E_OK	Normal completion
E_ID	Bad identifier
E_NOEXS	Object does not exist

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

This system call takes 64-bit stime_u and utime_u in microseconds instead of the return parameters stime and utime of td_inf_tsk.

The specification of this system call is same as that of td_inf_tsk, except that the return parameters are replaced with stime_u and utime_u. For more details, see the description of td_inf_tsk.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

6.1.43 td_get_reg - Get Task Register

C Language Interface

```
#include <tk/dbgspt.h>
```

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

Parameter

ID	tskid	Task ID	Target task ID (TSK_SELF cannot be specified)
T_REGS*	pk_regs	Packet of Registers	Pointer to the area to return the general register values
T_EIT*	pk_eit	Packet of EIT Registers	Pointer to the area to return the values of registers saved when an exception occurs
T_CREGS*	pk_cregs	Packet of Control Registers	Pointer to the area to return the control register values

Return Parameter

ER	ercd	Error Code	Error code

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

Error Code

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 a RUNNING state task)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

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 RUNNING state. Except when a task-independent portion is executing, the current RUNNING state task is the invoking task.

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

The contents of T_REGS, T_EIT, and T_CREGS are implementation-dependent.

6.1.44 td_set_reg - Set Task Registers

C Language Interface

```
#include <tk/dbgspt.h>
```

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

Parameter

ID	tskid	Task ID	Target task ID (TSK_SELF cannot be specified)
CONST T_REGS*	pk_regs	Packet of Registers	General registers
CONST T_EIT*	pk_eit	Packet of EIT Registers	Registers saved when EIT occurs
CONST 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 Parameter

ER	ercd	Error Code	Error code

Error Code

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 a RUNNING state task)

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

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 RUNNING state. Except when a task-independent portion is executing, the current RUNNING state task is the invoking task.

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

The contents of T_REGS, T_EIT, and T_CREGS are implementation-dependent.

6.1.45 td_get_tim - Get System Time

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_get_tim (SYSTIM *tim , UINT *ofs);

Parameter

SYSTIM*	tim	Time	Pointer to the area to return the current time (ms)
UINT*	ofs	Offset	Pointer to the area to return the return parameter ofs

Return Parameter

ER	ercd	Error Code	Error code
SYSTIM	tim	Time	Current time (in milliseconds)
UINT	ofs	Offset	Elapsed time from tim (nanoseconds)

tim Detail:

W	hi	High 32 bits	Higher 32 bits of current time of the system time
UW	lo	Low 32 bits	Lower 32 bits of current time of the system time

Error Code

E_OK	Normal completion

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

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 the resolution of timer interrupt intervals (cycles), but even more precise time information is obtained in ofs as the elapsed time from tim in nanoseconds. The resolution of ofs is implementation-dependent, but generally is the resolution of hardware timer.

Since tim is a cumulative 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 elapsed time from the previous timer interrupt is returned in ofs. Accordingly, in some cases ofs will be longer than the timer interrupt cycle. The length of elapsed time that can be measured by ofs depends on the hardware, but preferably it should be possible 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.

6.1.46 td_get_tim_u - Get System Time (Microseconds)

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_get_tim_u (SYSTIM_U *tim_u , UINT *ofs);

Parameter

SYSTIM_U*	tim_u	Time	Pointer to the area to return the current time (microseconds)
UINT*	ofs	Offset	Pointer to the area to return the return parameter ofs

Return Parameter

ER	ercd	Error Code	Error code
SYSTIM_U	tim_u	Time	Current time (in microseconds)
UINT	ofs	Offset	Elapsed time from tim_u (nanoseconds)

Error Code

E_OK	Normal completion

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

This system call takes 64-bit tim_u in microseconds instead of the return parameter tim of td_get_tim.

The specification of this system call is same as that of td_get_tim, except that the return parameter is replaced with tim_u. For more details, see the description of td_get_tim.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

6.1.47 td_get_otm - Get Operating Time

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_get_otm (SYSTIM *tim , UINT *ofs);

Parameter

SYSTIM*	tim	Time	Pointer to the area to return the operating time (ms)
UINT*	ofs	Offset	Pointer to the area to return the return parameter ofs

Return Parameter

ER	ercd	Error Code	Error code
SYSTIM	tim	Time	Operating time (ms)
UINT	ofs	Offset	Elapsed time from tim (nanoseconds)

tim Detail:

W	hi	High 32 bits	Higher 32 bits of the system operating time
UW	lo	Low 32 bits	Lower 32 bits of the system operating time

Error Code

E_OK	Normal completion

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

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 the resolution of timer interrupt intervals (cycles), but even more precise time information is obtained in ofs as the elapsed time from tim in nanoseconds. The resolution of ofs is implementation-dependent, but generally is the resolution of hardware timer.

Since tim is a cumulative 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 elapsed time from the previous timer interrupt is returned in ofs. Accordingly, in some cases ofs will be longer than the timer interrupt cycle. The length of elapsed time that can be measured by ofs depends on the hardware, but preferably it should be possible 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.

6.1.48 td_get_otm_u - Get Operating Time (Microseconds)

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_get_otm_u (SYSTIM_U *tim_u , UINT *ofs);

Parameter

SYSTIM_U*	tim_u	Time	Pointer to the area to return the operating time (microseconds)
UINT*	ofs	Offset	Pointer to the area to return the return parameter ofs

Return Parameter

ER	ercd	Error Code	Error code
SYSTIM_U	tim_u	Time	Operating time (microseconds)
UINT	ofs	Offset	Elapsed time from tim_u (nanoseconds)

Error Code

E_OK	Normal completion

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

This system call takes 64-bit tim_u in microseconds instead of the return parameter tim of td_get_otm.

The specification of this system call is same as that of td_get_otm, except that the return parameter is replaced with tim_u. For more details, see the description of td_get_otm.

Difference from T-Kernel 1.0

This system call was added in T-Kernel 2.0.

6.1.49 td_ref_dsname - Refer to DS Object Name

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_dsname (UINT type , ID id , UB *dsname);

Parameter

UINT	type	Object Type	Target object type
ID	id	Object ID	Object ID
UB*	dsname	DS Object Name	Pointer to the area to return the DS object name

Return Parameter

ER	ercd	Error Code	Error code

dsname Detail:
DS object name, set at object creation or by td_set_dsname()

Error Code

E_OK	Normal completion
E_PAR	Invalid object type
E_NOEXS	Object does not exist
E_OBJ	DS object name is not used

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

References the 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	Mailbox
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

DS object name is valid if TA_DSNAME is set as object attribute. If DS object name is changed by td_set_dsname(), then td_ref_dsname() references the new name.

DS object name needs to satisfy the following conditions:

Available characters (UB)
: a to z, A to Z, 0 to 9

Name length
: 8-byte (filled with NULL for shorter name)

However, character code range is not checked by T-Kernel.

6.1.50 td_set_dsname - Set DS Object Name

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_set_dsname (UINT type , ID id , CONST UB *dsname);

Parameter

UINT	type	Object Type	Target object type
ID	id	Object ID	Object ID
CONST UB*	dsname	DS Object Name	DS object name to be set

Return Parameter

ER	ercd	Error Code	Error code

Error Code

E_OK	Normal completion
E_PAR	Invalid object type
E_NOEXS	Object does not exist
E_OBJ	DS object name is not used

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Re-sets 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() .

DS object name needs to satisfy the following conditions:

Available characters (UB)
: a to z, A to Z, 0 to 9

Name length
: 8-byte (filled with NULL for shorter name)

However, character code range is not checked by T-Kernel.

DS object name is valid if TA_DSNAME is set as object attribute. td_set_dsname() returns E_OBJ error if TA_DSNAME attribute is not specified.

6.2 Trace Functions

Trace functions are functions for enabling a debugger to trace program execution. Execution trace is performed by setting hook routines.

- Return from a hook routine must be made 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 C language function saving rules.
- In a hook routine, limitations on states must not be loosened 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. Using too much stack 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 is a safer option.

6.2.1 td_hok_svc - Define System Call/Extended SVC Hook Routine

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_hok_svc (CONST TD_HSVC *hsvc);

Parameter

CONST TD_HSVC*	hsvc	SVC Hook Routine	Hook routine definition information

hsvc Detail:

FP	enter	Hook Routine before Calling	Hook routine before calling
FP	leave	Hook Routine after Calling	Hook routine after calling

Return Parameter

ER	ercd	Error Code	Error code

Error Code

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

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 an object of a trace.

T-Kernel/DS system calls (td_???_???) are not objects of a trace.

A hook routine runs as a quasi-task portion of the task that called a system call or extended SVC for which a hook routine is set. Therefore, 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.

void *enter (FN fncd , TD_CALINF *calinf , ...);

FN	fncd	Function Codes< 0 System call ≧ 0 Extended SVC
TD_CALINF*	calinf	Caller information
	...	Parameters (variable number)
Return		Any value passed to leave()

```
typedef struct td_calinf {
      Information to determine the caller for the system call or extended SVC;
      it is preferable to include the information for the stack back-trace.
      The contents are implementation-dependent,
      but generally consist of register values such as stack pointer and program counter.
} TD_CALINF;
```

enter is called right before a system call or extended SVC.

The value passed in the return code is passed transparently to the corresponding leave(). This makes it possible to pair 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)
```

- For system call

  The parameters are the same as the system call parameters.

Example 6.1 tk_wai_sem(ID semid, INT cnt, TMO tmout)

```
enter(TFN_WAI_SEM, &calinf, semid, cnt, tmout)
```

- For extended SVC

  The parameters are as in the packet passed to the extended SVC handler.

  fncd is likewise the same as that passed to the extended SVC handler.

  enter (FN fncd , TD_CALINF *calinf , void *pk_para);
  void leave (FN fncd , INT ret , void *exinf);

FN	fncd	Function Codes
INT	ret	Return code of the system call or extended SVC
void*	exinf	Any value returned by enter()

enter 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 leave() only 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().

6.2.2 td_hok_dsp - Define Task Dispatch Hook Routine

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_hok_dsp (CONST TD_HDSP *hdsp);

Parameter

CONST TD_HDSP*	hdsp	Dispatcher Hook Routine	Hook routine definition information

hdsp Detail:

FP	exec	Hook Routine when Execution Starts	Hook routine when execution starts
FP	stop	Hook Routine when Execution Stops	Hook routine when execution stops

Return Parameter

ER	ercd	Error Code	Error code

Error Code

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Sets hook routines in the task dispatcher. Setting NULL in hdsp cancels a hook routine.

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 system calls (td_...) may be invoked.

void exec (ID tskid , INT lsid);

ID	tskid	Task ID of the started or resumed task
INT	lsid	Logical space ID of the task designated in tskid

exec is called when the designated task starts execution or resumes. At the time exec() is called, the task designated in tskid is already in RUNNING 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);

ID	tskid	Task ID of the executed or stopped task
INT	lsid	Logical space ID of the task designated in tskid
UINT	tskstat	State of the task designated in tskid

stop is called when the designated task executes or stops. tskstat indicates the task state after stopping, as one of the following states:

TTS_RDY	READY state
TTS_WAI	WAITING state
TTS_SUS	SUSPENDED state
TTS_WAS	WAITING-SUSPENDED 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.

6.2.3 td_hok_int - Define Interrupt Handler Hook Routine

C Language Interface

```
#include <tk/dbgspt.h>
```

ER ercd = td_hok_int (CONST TD_HINT *hint);

Parameter

CONST TD_HINT*	hint	Interrupt Handler Hook Routine	Hook routine definition information

hint Detail:

FP	enter	Hook Routine before Calling Handler	Hook routine before calling handler
FP	leave	Hook Routine after Calling Handler	Hook routine after calling handler

Return Parameter

ER	ercd	Error Code	Error code

Error Code

None

Valid Context

Task portion	Quasi-task portion	Task-independent portion
YES	YES	YES

Description

Sets hook routines before and after an interrupt handler is called. Hook routine setting cannot be done individually 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);

UINT	dintno	Interrupt handler 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().

Chapter 7

Appendix

7.1 Specification Related to Device Drivers to be Used as Reference

In this section, the specifications related to the device management functions or device drivers in the T-Kernel/SM that are not included in the latest specification of T-Kernel though described in the device management functions in the T-Kernel specification Ver.1.00.xx and for which implementation examples are available on the existing systems are described.

It is preferable to examine the description described in this section when the compatibility with the device related functions of the existing T-Kernel application system or the device drivers for the existing T-Kernel need to be considered.

Details and the latest information of the specification, and the operation method of its specification need to be confirmed separately.

7.1.1 Disk Kind for Device Attributes

In the definition of device attributes in the T-Kernel specification Ver.1.00.xx, the following disk kinds are defined:

```
/* 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 */
```

7.1.2 Device Attribute Data

`TDN_DISKINFO`
: Disk information

  The definition of DiskFormat that is used in the attribute data in the device common attribute in the T-Kernel specification Ver.1.00.xx. DiskFormat is included in the data type DiskInfo that is used in the disk information (`TDN_DISKINFO`) in the attribute data.

```
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 ;
```

`TDN_DISPSPEC`
: Display Device Specification

  The definition of DEV_SPEC that is used in the attribute data in the device common attribute in the T-Kernel specification Ver.1.00.xx. DEV_SPEC is a data type that is used in the display device specification (`TDN_DISPSPEC`) in the attribute data.

```
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;
```

7.1.3 Event Type of the Device Event Notification

The following event types are defined in the device event notification in the T-Kernel specification Ver.1.00.xx:

```
typedef enum tdevttyp {
        TDE_unknown     = 0,            /* undefined */
        TDE_MOUNT       = 0x01,         /* media insert */
        TDE_EJECT       = 0x02,         /* Eject media */
        TDE_ILLMOUNT    = 0x03,         /* illegal media insert */
        TDE_ILLEJECT    = 0x04,         /* illegal media eject */
        TDE_REMOUNT     = 0x05,         /* media reinsert */
        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 position 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          /* autopower on notify */
} TDEvtTyp;
```

Chapter 8

Reference

8.1 List of C Language Interface

8.1.1 T-Kernel/OS

8.1.1.1 Task Management Functions

- ID tskid = tk_cre_tsk (CONST T_CTSK *pk_ctsk);
- ER ercd = tk_del_tsk (ID tskid);
- ER ercd = tk_sta_tsk (ID tskid, INT stacd);
- void tk_ext_tsk (void);
- void tk_exd_tsk (void);
- ER ercd = tk_ter_tsk (ID tskid);
- ER ercd = tk_chg_pri (ID tskid, PRI tskpri);
- ER ercd = tk_chg_slt (ID tskid, RELTIM slicetime);
- ER ercd = tk_chg_slt_u (ID tskid, RELTIM_U slicetime_u);
- ER ercd = tk_get_tsp (ID tskid, T_TSKSPC *pk_tskspc);
- ER ercd = tk_set_tsp (ID tskid, CONST T_TSKSPC *pk_tskspc);
- ID resid = tk_get_rid (ID tskid);
- ID oldid = tk_set_rid (ID tskid, ID resid);
- ER ercd = tk_get_reg (ID tskid, T_REGS *pk_regs, T_EIT *pk_eit, T_CREGS *pk_cregs);
- ER ercd = tk_set_reg (ID tskid, CONST T_REGS *pk_regs, CONST T_EIT *pk_eit, CONST T_CREGS *pk_cregs);
- ER ercd = tk_get_cpr (ID tskid, INT copno, T_COPREGS *pk_copregs);
- ER ercd = tk_set_cpr (ID tskid, INT copno, CONST T_COPREGS *pk_copregs);
- ER ercd = tk_inf_tsk (ID tskid, T_ITSK *pk_itsk, BOOL clr);
- ER ercd = tk_inf_tsk_u (ID tskid, T_ITSK_U *pk_itsk_u, BOOL clr);
- ER ercd = tk_ref_tsk (ID tskid, T_RTSK *pk_rtsk);
- ER ercd = tk_ref_tsk_u (ID tskid, T_RTSK_U *pk_rtsk_u);

8.1.1.2 Task Synchronization Functions

- ER ercd = tk_slp_tsk (TMO tmout);
- ER ercd = tk_slp_tsk_u (TMO_U tmout_u);
- ER ercd = tk_wup_tsk (ID tskid);
- INT wupcnt = tk_can_wup (ID tskid);
- ER ercd = tk_rel_wai (ID tskid);
- ER ercd = tk_sus_tsk (ID tskid);
- ER ercd = tk_rsm_tsk (ID tskid);

- ER ercd = tk_frsm_tsk (ID tskid);
- ER ercd = tk_dly_tsk (RELTIM dlytim);
- ER ercd = tk_dly_tsk_u (RELTIM_U dlytim_u);
- ER ercd = tk_sig_tev (ID tskid, INT tskevt);
- INT tevptn = tk_wai_tev (INT waiptn, TMO tmout);
- INT tevptn = tk_wai_tev_u (INT waiptn, TMO_U tmout_u);
- INT tskwait = tk_dis_wai (ID tskid, UINT waitmask);
- ER ercd = tk_ena_wai (ID tskid);

8.1.1.3 Task Exception Handling Functions

- ER ercd = tk_def_tex (ID tskid, CONST T_DTEX *pk_dtex);
- ER ercd = tk_ena_tex (ID tskid, UINT texptn);
- ER ercd = tk_dis_tex (ID tskid, UINT texptn);
- ER ercd = tk_ras_tex (ID tskid, INT texcd);
- INT texcd = tk_end_tex (BOOL enatex);
- ER ercd = tk_ref_tex (ID tskid, T_RTEX *pk_rtex);

8.1.1.4 Synchronization and Communication Functions

- ID semid = tk_cre_sem (CONST T_CSEM *pk_csem);
- ER ercd = tk_del_sem (ID semid);
- ER ercd = tk_sig_sem (ID semid, INT cnt);
- ER ercd = tk_wai_sem (ID semid, INT cnt, TMO tmout);
- ER ercd = tk_wai_sem_u (ID semid, INT cnt, TMO_U tmout_u);
- ER ercd = tk_ref_sem (ID semid, T_RSEM *pk_rsem);
- ID flgid = tk_cre_flg (CONST T_CFLG *pk_cflg);
- ER ercd = tk_del_flg (ID flgid);
- ER ercd = tk_set_flg (ID flgid, UINT setptn);
- ER ercd = tk_clr_flg (ID flgid, UINT clrptn);
- ER ercd = tk_wai_flg (ID flgid, UINT waiptn, UINT wfmode, UINT *p_flgptn, TMO tmout);
- ER ercd = tk_wai_flg_u (ID flgid, UINT waiptn, UINT wfmode, UINT *p_flgptn, TMO_U tmout_u);
- ER ercd = tk_ref_flg (ID flgid, T_RFLG *pk_rflg);
- ID mbxid = tk_cre_mbx (CONST T_CMBX* pk_cmbx);
- ER ercd = tk_del_mbx (ID mbxid);
- ER ercd = tk_snd_mbx (ID mbxid, T_MSG *pk_msg);
- ER ercd = tk_rcv_mbx (ID mbxid, T_MSG **ppk_msg, TMO tmout);
- ER ercd = tk_rcv_mbx_u (ID mbxid, T_MSG **ppk_msg, TMO_U tmout_u);
- ER ercd = tk_ref_mbx (ID mbxid, T_RMBX *pk_rmbx);

8.1.1.5 Extended Synchronization and Communication Functions

- ID mtxid = tk_cre_mtx (CONST T_CMTX *pk_cmtx);
- ER ercd = tk_del_mtx (ID mtxid);
- ER ercd = tk_loc_mtx (ID mtxid, TMO tmout);
- ER ercd = tk_loc_mtx_u (ID mtxid, TMO_U tmout_u);
- ER ercd = tk_unl_mtx (ID mtxid);
- ER ercd = tk_ref_mtx (ID mtxid, T_RMTX *pk_rmtx);
- ID mbfid = tk_cre_mbf (CONST T_CMBF *pk_cmbf);
- ER ercd = tk_del_mbf (ID mbfid);
- ER ercd = tk_snd_mbf (ID mbfid, CONST void *msg, INT msgsz, TMO tmout);
- ER ercd = tk_snd_mbf_u (ID mbfid, CONST void *msg, INT msgsz, TMO_U tmout_u);
- INT msgsz = tk_rcv_mbf (ID mbfid, void *msg, TMO tmout);
- INT msgsz = tk_rcv_mbf_u (ID mbfid, void *msg, TMO_U tmout_u);
- ER ercd = tk_ref_mbf (ID mbfid, T_RMBF *pk_rmbf);
- ID porid = tk_cre_por (CONST T_CPOR *pk_cpor);
- ER ercd = tk_del_por (ID porid);
- INT rmsgsz = tk_cal_por (ID porid, UINT calptn, void *msg, INT cmsgsz, TMO tmout);
- INT rmsgsz = tk_cal_por_u (ID porid, UINT calptn, void *msg, INT cmsgsz, TMO_U tmout_u);
- INT cmsgsz = tk_acp_por (ID porid, UINT acpptn, RNO *p_rdvno, void *msg, TMO tmout);
- INT cmsgsz = tk_acp_por_u (ID porid, UINT acpptn, RNO *p_rdvno, void *msg, TMO_U tmout_u);
- ER ercd = tk_fwd_por (ID porid, UINT calptn, RNO rdvno, CONST void *msg, INT cmsgsz);
- ER ercd = tk_rpl_rdv (RNO rdvno, CONST void *msg, INT rmsgsz);
- ER ercd = tk_ref_por (ID porid, T_RPOR *pk_rpor);

8.1.1.6 Memory Pool Management Functions

- ID mpfid = tk_cre_mpf (CONST T_CMPF *pk_cmpf);
- ER ercd = tk_del_mpf (ID mpfid);
- ER ercd = tk_get_mpf (ID mpfid, void **p_blf, TMO tmout);
- ER ercd = tk_get_mpf_u (ID mpfid, void **p_blf, TMO_U tmout_u);
- ER ercd = tk_rel_mpf (ID mpfid, void *blf);
- ER ercd = tk_ref_mpf (ID mpfid, T_RMPF *pk_rmpf);
- ID mplid = tk_cre_mpl (CONST T_CMPL *pk_cmpl);
- ER ercd = tk_del_mpl (ID mplid);
- ER ercd = tk_get_mpl (ID mplid, INT blksz, void **p_blk, TMO tmout);
- ER ercd = tk_get_mpl_u (ID mplid, INT blksz, void **p_blk, TMO_U tmout_u);
- ER ercd = tk_rel_mpl (ID mplid, void *blk);
- ER ercd = tk_ref_mpl (ID mplid, T_RMPL *pk_rmpl);

8.1.1.7 Time Management Functions

- ER ercd = tk_set_tim (CONST SYSTIM *pk_tim);
- ER ercd = tk_set_tim_u (SYSTIM_U tim_u);
- ER ercd = tk_get_tim (SYSTIM *pk_tim);
- ER ercd = tk_get_tim_u (SYSTIM_U *tim_u, UINT *ofs);
- ER ercd = tk_get_otm (SYSTIM *pk_tim);
- ER ercd = tk_get_otm_u (SYSTIM_U *tim_u, UINT *ofs);
- ID cycid = tk_cre_cyc (CONST T_CCYC *pk_ccyc);
- ID cycid = tk_cre_cyc_u (CONST T_CCYC_U *pk_ccyc_u);
- ER ercd = tk_del_cyc (ID cycid);
- ER ercd = tk_sta_cyc (ID cycid);
- ER ercd = tk_stp_cyc (ID cycid);
- ER ercd = tk_ref_cyc (ID cycid, T_RCYC *pk_rcyc);
- ER ercd = tk_ref_cyc_u (ID cycid, T_RCYC_U *pk_rcyc_u);
- ID almid = tk_cre_alm (CONST T_CALM *pk_calm);
- ER ercd = tk_del_alm (ID almid);
- ER ercd = tk_sta_alm (ID almid, RELTIM almtim);
- ER ercd = tk_sta_alm_u (ID almid, RELTIM_U almtim_u);
- ER ercd = tk_stp_alm (ID almid);
- ER ercd = tk_ref_alm (ID almid, T_RALM *pk_ralm);
- ER ercd = tk_ref_alm_u (ID almid, T_RALM_U *pk_ralm_u);

8.1.1.8 Interrupt Management Functions

- ER ercd = tk_def_int (UINT dintno, CONST T_DINT *pk_dint);
- void tk_ret_int (void);

8.1.1.9 System Management Functions

- ER ercd = tk_rot_rdq (PRI tskpri);
- ID tskid = tk_get_tid (void);
- ER ercd = tk_dis_dsp (void);
- ER ercd = tk_ena_dsp (void);
- ER ercd = tk_ref_sys (T_RSYS *pk_rsys);
- ER ercd = tk_set_pow (UINT powmode);
- ER ercd = tk_ref_ver (T_RVER *pk_rver);

8.1.1.10 Subsystem Management Functions

- ER ercd = tk_def_ssy (ID ssid, CONST T_DSSY *pk_dssy);
- ER ercd = tk_sta_ssy (ID ssid, ID resid, INT info);
- ER ercd = tk_cln_ssy (ID ssid, ID resid, INT info);
- ER ercd = tk_evt_ssy (ID ssid, INT evttyp, ID resid, INT info);
- ER ercd = tk_ref_ssy (ID ssid, T_RSSY *pk_rssy);
- ER ercd = tk_cre_res (void);
- ER ercd = tk_del_res (ID resid);
- ER ercd = tk_get_res (ID resid, ID ssid, void **p_resblk);

8.1.2 T-Kernel/SM

8.1.2.1 System Memory Management Functions

- ER ercd = tk_get_smb (void **addr, INT nblk, UINT attr);
- ER ercd = tk_rel_smb (void *addr);
- ER ercd = tk_ref_smb (T_RSMB *pk_rsmb);
- 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);

8.1.2.2 Address Space Management Functions

- ER ercd = SetTaskSpace (ID tskid);
- ER ercd = ChkSpaceR (CONST void *addr, INT len);
- ER ercd = ChkSpaceRW (CONST void *addr, INT len);
- ER ercd = ChkSpaceRE (CONST void *addr, INT len);
- INT rlen = ChkSpaceBstrR (CONST UB *str, INT max);
- INT rlen = ChkSpaceBstrRW (CONST UB *str, INT max);
- INT rlen = ChkSpaceTstrR (CONST TC *str, INT max);
- INT rlen = ChkSpaceTstrRW (CONST TC *str, INT max);
- ER ercd = LockSpace (CONST void *addr, INT len);

- ER ercd = UnlockSpace (CONST void *addr, INT len);
- INT rlen = CnvPhysicalAddr (CONST void *vaddr, INT len, void **paddr);
- ER ercd = MapMemory (CONST void *paddr, INT len, UINT attr, void **laddr);
- ER ercd = UnmapMemory (CONST void *laddr);
- ER ercd = GetSpaceInfo (CONST void *addr, INT len, T_SPINFO *pk_spinfo);
- INT rlen = SetMemoryAccess (CONST void *addr, INT len, UINT mode);

8.1.2.3 Device Management Functions

- ID dd = tk_opn_dev (CONST UB *devnm, UINT omode);
- ER ercd = tk_cls_dev (ID dd, UINT option);
- ID reqid = tk_rea_dev (ID dd, W start, void *buf, W size, TMO tmout);
- ID reqid = tk_rea_dev_du (ID dd, D start_d, void *buf, W size, TMO_U tmout_u);
- ER ercd = tk_srea_dev (ID dd, W start, void *buf, W size, W *asize);
- ER ercd = tk_srea_dev_d (ID dd, D start_d, void *buf, W size, W *asize);
- ID reqid = tk_wri_dev (ID dd, W start, CONST void *buf, W size, TMO tmout);
- ID reqid = tk_wri_dev_du (ID dd, D start_d, CONST void *buf, W size, TMO_U tmout_u);
- ER ercd = tk_swri_dev (ID dd, W start, CONST void *buf, W size, W *asize);
- ER ercd = tk_swri_dev_d (ID dd, D start_d, CONST void *buf, W size, W *asize);
- ID creqid = tk_wai_dev (ID dd, ID reqid, W *asize, ER *ioer, TMO tmout);
- ID creqid = tk_wai_dev_u (ID dd, ID reqid, W *asize, ER *ioer, TMO_U tmout_u);
- INT dissus = tk_sus_dev (UINT mode);
- ID pdevid = tk_get_dev (ID devid, UB *devnm);
- ID devid = tk_ref_dev (CONST UB *devnm, T_RDEV *rdev);
- ID devid = tk_oref_dev (ID dd, T_RDEV *rdev);
- INT remcnt = tk_lst_dev (T_LDEV *ldev, INT start, INT ndev);
- INT retcode = tk_evt_dev (ID devid, INT evttyp, void *evtinf);
- ID devid = tk_def_dev (CONST UB *devnm, CONST T_DDEV *ddev, T_IDEV *idev);
- ER ercd = tk_ref_idv (T_IDEV *idev);
- ER ercd = openfn (IDdevid, UINTomode, void * exinf);
- ER ercd = closefn (IDdevid, UINToption, void * exinf);
- ER ercd = execfn (T_DEVREQ * devreq, TMOtmout, void * exinf);
- ER ercd = execfn (T_DEVREQ_D * devreq_d, TMOtmout, void * exinf);
- ER ercd = execfn (T_DEVREQ * devreq, TMO_Utmout_u, void * exinf);
- ER ercd = execfn (T_DEVREQ_D * devreq_d, TMO_Utmout_u, void * exinf);
- INT creqno = waitfn (T_DEVREQ * devreq, INTnreq, TMOtmout * exinf);

- INT creqno = waitfn (T_DEVREQ_D * devreq_d, INTnreq, TMOtmout * exinf);
- INT creqno = waitfn (T_DEVREQ * devreq, INTnreq, TMO_Utmout_u * exinf);
- INT creqno = waitfn (T_DEVREQ_D * devreq_d, INTnreq, TMO_Utmout_u * exinf);
- ER ercd = abortfn (IDtskid, T_DEVRQ * devreq, INTnreq, void * exinf);
- ER ercd = abortfn (IDtskid, T_DEVRQ_D * devreq_d, INTnreq, void * exinf);
- INT retcode = eventfn (INTevttyp, void * evtinf, void * exinf);

8.1.2.4 Interrupt Management Functions

- DI (UINT intsts);
- EI (UINT intsts);
- BOOL disint = isDI (UINT intsts);
- UINT dintno = DINTNO (INTVEC intvec);
- void EnableInt (INTVEC intvec);
- void EnableInt (INTVEC intvec, INT level);
- void DisableInt (INTVEC intvec);
- void ClearInt (INTVEC intvec);
- void EndOfInt (INTVEC intvec);
- BOOL rasint = CheckInt (INTVEC intvec);
- void SetIntMode (INTVEC intvec, UINT mode);

8.1.2.5 I/O Port Access Support Functions

- void out_b (INT port, UB data);
- void out_h (INT port, UH data);
- void out_w (INT port, UW data);
- void out_d (INT port, UD data);
- UB data = in_b (INT port);
- UH data = in_h (INT port);
- UW data = in_w (INT port);
- UD data = in_d (INT port);
- void WaitUsec (UINT usec);
- void WaitNsec (UINT nsec);

8.1.2.6 Power Management Functions

- void low_pow (void);
- void off_pow (void);

8.1.2.7 System Configuration Information Management Functions

- INT ct = tk_get_cfn (CONST UB *name, INT *val, INT max);
- INT rlen = tk_get_cfs (CONST UB *name, UB *buf, INT max);

8.1.2.8 Memory Cache Control Functions

- INT rlen = SetCacheMode (void *addr, INT len, UINT mode);
- INT rlen = ControlCache (void *addr, INT len, UINT mode);

8.1.2.9 Physical Timer Functions

- ER ercd = StartPhysicalTimer (UINT ptmrno, UW limit, UINT mode);
- ER ercd = StopPhysicalTimer (UINT ptmrno);
- ER ercd = GetPhysicalTimerCount (UINT ptmrno, UW *p_count);
- ER ercd = DefinePhysicalTimerHandler (UINT ptmrno, CONST T_DPTMR *pk_dptmr);
- ER ercd = GetPhysicalTimerConfig (UINT ptmrno, T_RPTMR *pk_rptmr);

8.1.2.10 Utility Functions

- void SetOBJNAME (void *exinf, CONST UB *name);
- ER ercd = CreateLock (FastLock *lock, CONST UB *name);
- void DeleteLock (FastLock *lock);
- void Lock (FastLock *lock);
- void Unlock (FastLock *lock);
- ER ercd = CreateMLock (FastMLock *lock, CONST UB *name);
- ER ercd = DeleteMLock (FastMLock *lock);
- ER ercd = MLock (FastMLock *lock, INT no);
- ER ercd = MLockTmo (FastMLock *lock, INT no, TMO tmout);
- ER ercd = MLockTmo_u (FastMLock *lock, INT no, TMO_U tmout_u);
- ER ercd = MUnlock (FastMLock *lock, INT no);

8.1.3 T-Kernel/DS

8.1.3.1 Kernel Internal State Acquisition Functions

- INT ct = td_lst_tsk (ID list[], INT nent);
- INT ct = td_lst_sem (ID list[], INT nent);
- INT ct = td_lst_flg (ID list[], INT nent);
- INT ct = td_lst_mbx (ID list[], INT nent);
- INT ct = td_lst_mtx (ID list[], INT nent);
- INT ct = td_lst_mbf (ID list[], INT nent);
- INT ct = td_lst_por (ID list[], INT nent);
- INT ct = td_lst_mpf (ID list[], INT nent);
- INT ct = td_lst_mpl (ID list[], INT nent);
- INT ct = td_lst_cyc (ID list[], INT nent);
- INT ct = td_lst_alm (ID list[], INT nent);
- INT ct = td_lst_ssy (ID list[], INT nent);
- INT ct = td_rdy_que (PRI pri, ID list[], INT nent);
- INT ct = td_sem_que (ID semid, ID list[], INT nent);
- INT ct = td_flg_que (ID flgid, ID list[], INT nent);
- INT ct = td_mbx_que (ID mbxid, ID list[], INT nent);
- INT ct = td_mtx_que (ID mtxid, ID list[], INT nent);
- INT ct = td_smbf_que (ID mbfid, ID list[], INT nent);
- INT ct = td_rmbf_que (ID mbfid, ID list[], INT nent);

- INT ct = td_cal_que (ID porid, ID list[], INT nent);
- INT ct = td_acp_que (ID porid, ID list[], INT nent);
- INT ct = td_mpf_que (ID mpfid, ID list[], INT nent);
- INT ct = td_mpl_que (ID mplid, ID list[], INT nent);
- ER ercd = td_ref_tsk (ID tskid, TD_RTSK *rtsk);
- ER ercd = td_ref_tsk_u (ID tskid, TD_RTSK_U *rtsk_u);
- ER ercd = td_ref_tex (ID tskid, TD_RTEX *pk_rtex);
- ER ercd = td_ref_sem (ID semid, TD_RSEM *rsem);
- ER ercd = td_ref_flg (ID flgid, TD_RFLG *rflg);
- ER ercd = td_ref_mbx (ID mbxid, TD_RMBX *rmbx);
- ER ercd = td_ref_mtx (ID mtxid, TD_RMTX *rmtx);
- ER ercd = td_ref_mbf (ID mbfid, TD_RMBF *rmbf);
- ER ercd = td_ref_por (ID porid, TD_RPOR *rpor);
- ER ercd = td_ref_mpf (ID mpfid, TD_RMPF *rmpf);
- ER ercd = td_ref_mpl (ID mplid, TD_RMPL *rmpl);
- ER ercd = td_ref_cyc (ID cycid, TD_RCYC *rcyc);
- ER ercd = td_ref_cyc_u (ID cycid, TD_RCYC_U *rcyc_u);
- ER ercd = td_ref_alm (ID almid, TD_RALM *ralm);
- ER ercd = td_ref_alm_u (ID almid, TD_RALM_U *ralm_u);
- ER ercd = td_ref_sys (TD_RSYS *pk_rsys);
- ER ercd = td_ref_ssy (ID ssid, TD_RSSY *rssy);
- ER ercd = td_inf_tsk (ID tskid, TD_ITSK *pk_itsk, BOOL clr);
- ER ercd = td_inf_tsk_u (ID tskid, TD_ITSK_U *itsk_u, BOOL clr);
- ER ercd = td_get_reg (ID tskid, T_REGS *pk_regs, T_EIT *pk_eit, T_CREGS *pk_cregs);
- ER ercd = td_set_reg (ID tskid, CONST T_REGS *pk_regs, CONST T_EIT *pk_eit, CONST T_CREGS *pk_cregs);
- ER ercd = td_get_tim (SYSTIM *tim, UINT *ofs);
- ER ercd = td_get_tim_u (SYSTIM_U *tim_u, UINT *ofs);
- ER ercd = td_get_otm (SYSTIM *tim, UINT *ofs);
- ER ercd = td_get_otm_u (SYSTIM_U *tim_u, UINT *ofs);
- ER ercd = td_ref_dsname (UINT type, ID id, UB *dsname);
- ER ercd = td_set_dsname (UINT type, ID id, CONST UB *dsname);

8.1.3.2 Trace Functions

- ER ercd = td_hok_svc (CONST TD_HSVC *hsvc);
- ER ercd = td_hok_dsp (CONST TD_HDSP *hdsp);
- ER ercd = td_hok_int (CONST TD_HINT *hint);

8.2 List of Error Codes

8.2.1 Normal Completion Error Class (0)

Error code name	Error Codes	Summary description
E_OK	0	Normal completion

8.2.2 Normal completion Internal Error Class (5 to 8)

Error code name	Error Codes	Summary description
E_SYS	ERCD(-5, 0)	System error

An error of unknown cause affecting the system as a whole.

Error code name	Error Codes	Summary description
E_NOCOP	ERCD(-6, 0)	Unavailable co-processor

This error code is returned when the specified co-processor is not installed in the currently running hardware, or abnormal co-processor condition was detected.

8.2.3 Unsupported Error Class (9 to 16)

Error code name	Error Codes	Summary description
E_NOSPT	ERCD(-9, 0)	Unsupported function

When some system call functions are not supported and such a function is invoked, error code E_RSATR or E_NOSPT is returned. If E_RSATR does not apply, error code E_NOSPT is returned.

Error code name	Error Codes	Summary description
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 (a positive function code).

Error code name	Error Codes	Summary description
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.

8.2.4 Parameter Error Class (17 to 24)

Error code name	Error Codes	Summary description
E_PAR	ERCD(-17, 0)	Parameter error

Checking for this error may be omitted if system-dependent optimization is implemented.

Error code name	Error Codes	Summary description
E_ID	ERCD(-18, 0)	Invalid ID number

E_ID is an error that is returned only for objects having an ID number.

Error code E_PAR is returned when a static error is detected for such as reserved number or out of range in the case of interrupt handler number.

8.2.5 Call Context Error Class (25 to 32)

Error code name	Error Codes	Summary description
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 RUNNING state).

This error must be returned whenever there is a semantic 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 WAITING 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.

Error code name	Error Codes	Summary description
E_MACV	ERCD(-26, 0)	Memory cannot be accessed; memory access privilege error

Error detection is implementation-dependent.

Error code name	Error Codes	Summary description
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.

Error code name	Error Codes	Summary description
E_ILUSE	ERCD(-28, 0)	System call illegal use

8.2.6 Resource Constraint Error Class (33 to 40)

Error code name	Error Codes	Summary description
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 area, memory pool area, message buffer area or the like.

Error code name	Error Codes	Summary description
E_LIMIT	ERCD(-34, 0)	System limit exceeded

This error code is returned, for example, when it is attempted to create more object(s) than the system allows.

8.2.7 Object State Error Class (41 to 48)

Error code name	Error Codes	Summary description
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

8.2.8 Wait Error Class (49 to 56)

Error code name	Error Codes	Summary description
E_RLWAI	ERCD(-49, 0)	Waiting state was forcibly released
E_TMOUT	ERCD(-50, 0)	Polling failed or timeout
E_DLT	ERCD(-51, 0)	Waiting object was deleted
E_DISWAI	ERCD(-52, 0)	Wait released due to disabling of wait

8.2.9 Device Error Class (57 to 64) (T-Kernel/SM)

Error code name	Error Codes	Summary description
E_IO	ERCD(-57, 0)	I/O error

※ Error information specific to individual devices may be defined in E_IO sub-codes.

Error code name	Error Codes	Summary description
E_NOMDA	ERCD(-58, 0)	No media

8.2.10 Status Error Class (65 to 72) (T-Kernel/SM)

Error code name	Error Codes	Summary description
E_BUSY	ERCD(-65, 0)	Busy
E_ABORT	ERCD(-66, 0)	Processing was aborted
E_RONLY	ERCD(-67, 0)	Write protected