TEF020-S004-01.01.01/en

μT-Kernel Specification

μT-Kernel 1.01.01

TEF020-S004-01.01.01/en

µT-Kernel Specification Version 1.01.01

TRON Architecture: Designed by Ken Sakamura

Contents

List of Figures

List of Tables

Change List

Version	Date	Page	Updated Content	Note
01.01.01	July 17, 2012	ix	Update List of Figures	
		x	Update List of Tables	
		115	[OLD] Figure 4.8 → [NEW] Figure 4.6	
		167	[OLD] Figure 4.9 → [NEW] Figure 4.7	
		167	[OLD] Figure 4.10 → [NEW] Figure 4.8	
		173	[OLD] Figure 4.11 → [NEW] Figure 4.9	
		174	[OLD] Figure 4.12 → [NEW] Figure 4.10	
		174	[OLD] Figure 4.13 → [NEW] Figure 4.11	
		181	[OLD] Figure 4.14 → [NEW] Figure 4.12	
		185	Table 4.4 now has the missed title.	

System Call Notation

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

`System call name`

Summary description

Summary description

[C Language Interface]

Indicates the C language interface for invoking the system call.

[Parameters]

Describes the system call parameters, i.e., the information passed to the OS when the system call is executed.

[Return Parameters]

Describes the system call return parameters, i.e., the information returned by the OS when execution of the system call ends.

[Error Codes]

Describes the errors that can be returned by the system call.

The following error codes are common to all system calls and are not included in the error code listings for individual system calls.

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

[Description]

Describes the system call functions.

When the values to be passed in a parameter are selected from various choices, the following notation is used in the parameter descriptions.

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

Example:

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

TWF_ANDW

TWF_ORW

(TWF_ANDW | TWF_CLR)

(TWF_ORW | TWF_CLR)

[Additional Notes]

Supplements the description by noting matters that need special attention or caution, etc.

[Rationale for the Specification]

Explains the reason for adopting a particular specification.

[Difference with T-Kernel]

Explains the difference in the specification between the µT-Kernel and T-Kernel.

[Difference with T-Kernel 1.00.00]

Explains the difference in the specification between the µT-Kernel and T-Kernel 1.00.00, which is the version before data types were changed.

[Porting Guideline]

Provides guidelines and considerations for porting between different µT-Kernel implementations and when porting from µT-Kernel to T-Kernel and vice versa.

Index of μT-Kernel API

The μT-Kernel APIs described in this specification are listed below in alphabetical order.

Chapter 1

µT-Kernel Overview

1.1 Positioning for µT-Kernel

µT-Kernel is a real-time OS for small scaled embedded systems, and as a target, it assumes 16-bit single chip microcomputers, environments with limited ROM/RAM, or systems without MMU, etc. So, this specification allows developers to easily optimize and adapt the installation on a small scaled embedded system. Also, taking compatibility with T-Kernel into account, µT-Kernel is designed to increase the distribution and portability of device drivers and middleware.

1.2 Background

Since the design of the T-Kernel specification, various attempts have been made to use the T-Kernel. There have also been attempts to apply the T-Kernel to small scaled embedded devices.

The T-Kernel source code is centrally managed by the T-Engine Forum to improve the distribution and portability of device drivers or middleware, and to prevent arbitrary distribution of different types of T-Kernel source code. However, when embedding into actual products, you may adapt the T-Kernel in accordance with the target system and remove any unnecessary functions. That is, you can use the changed T-Kernel as an object, but cannot distribute its source code.

As T-Kernel specification targets large scaled embedded systems, it includes advanced functions rarely used in small scaled embedded systems. Therefore, efforts towards optimization and adaptation have been made in conformity with the above constraints. As a result, when hardware with limited resources, such as single chip microcomputers, is targeted, it becomes clear that you cannot address it only by adapting the T-Kernel.

To address these issues without impairing the distribution and portability of device drivers or middleware, the µT-Kernel specification has been designed for small systems, based on the knowledge obtained by adapting the T-Kernel.

1.3 Design Policy

The design of the µT-Kernel specification is based on the policy of allowing easy optimization and adaptation of the µT-Kernel for small scaled embedded systems. However, taking compatibility with T-Kernel into account, it is also designed to improve the distribution and portability of device drivers or middleware.

For easy optimization and adaptation, functions not used in small systems and functions that add to the overheads of the whole system have been omitted. Functions for using resources effectively have been added. Also, unlike the T-Kernel, instead of retaining the unity of source code, the specification allows you to develop µT-Kernel in accordance with the requirements of the target systems.

For improving distribution and portability, similar to the T-Kernel, strong standardization of µT-Kernel ensures portability between different µT-Kernels. Further, unification of the interface with T-Kernel maintains high compatibility with T-Kernel. Specifically, the functions present in both µT-Kernel and T-Kernel are defined by the same interface, and the data type definitions in µT-Kernel are also common to T-Kernel. Therefore, using only the functions common to both µT-Kernel and T-Kernel, and writing programs according to the porting guidelines, allows you to port the program only by recompiling. A function present only in µT-Kernel may be a functional enhancement for T-Kernel, but it is a natural enhancement based on the T-Kernel format. So, when porting a program with that function to the T-Kernel, the program rarely needs to be modified.

Further, the purpose of including the device driver management system in the specification is to ensure distribution and portability. This allows you to create and utilize device drivers in the format common to T-Kernel, improving the distribution and portability of device drivers and applications.

1.4 Reference Code

As optimization and adaptation are particularly important to small scaled embedded systems, unlike T-Kernel, µT-Kernel does not retain the unity of source code. Instead, it provides a source code for reference, called "reference code".

The reference code is one implementation of the µT-Kernel, and is distributed by the T-Engine Forum. It is different from T-Kernel in that this reference code only represents µT-Kernel. OS implementors can modify it, or can implement it themselves. However, only those OS which behave in the same manner as the reference code are accepted as µT-Kernel specification OS.

The reference code defines detailed behaviors which are difficult to define in specifications, and deploying this reference code concept allows you to maintain the consistency of behaviors in various implementations, while promoting the optimization and adaptation in accordance with the targets.

1.5 Adaptation

µT-Kernel specification is designed by taking compatibility with T-Kernel into account. That is, when porting programs between µT-Kernel and T-Kernel, the specification is designed so that using only the functions common to them allows you to port programs only by recompiling and by minimum modification, even if modification is inevitable. Therefore, the µT-Kernel specification includes some functions that are unnecessary for some systems. However, if subset specifications are defined for such systems, the distribution and portability of device drivers or middleware, etc. will be impaired. Also, as necessary functions can vary according to the targets, it is difficult to define the subset specifications.

In µT-Kernel, specifications for creating subsets, such as level classification, are not defined. All OSs compliant with µT-Kernel specification should implement all functions. However, when incorporating µT-Kernel into target systems, you can omit unnecessary functions. That is, the OS provider should provide all functions included in the specification, but the user can select and use only the necessary functions.

Chapter 2

Concepts Underlying the µT-Kernel Specification

2.1 Meaning of Basic Terminology

2.1.1 Terminology about Implementation

1. **Implementation-defined**

   This item is not standardized in the µT-Kernel specification and should be defined for each implementation. The specifics of the implementation should be specified in the implementation specifications. In application programs, the portability for the portion dependent on implementation dependent items is not ensured.

2. **Implementation-dependent**

   This item shall indicate that in the µT-Kernel specification, the behavior varies according to the target system or system operating condition. The behavior should be defined for each implementation. The specifics of the implementation should be specified in the implementation specifications. In application programs, the portion dependent on implementation-dependent items essentially needs to be modified when porting.

2. **Implementation Specifications**

   This document explains the specifics of the implementation, etc. when the µT-Kernel is incorporated in the target system. Creation of implementation specifications is strongly recommended when the µT-Kernel is implemented, adapted and optimized for a target system. You need not name the target document "Implementation Specifications".

2.1.2 Meaning of Other Basic Terms

1. **Task, invoking task**

   The basic logical unit of concurrent program execution is called a “task”. While instructions within one task are executed in sequence, instructions within different tasks can be executed in parallel. This concurrent processing is a conceptual phenomenon from the standpoint of applications; in reality, 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”.

2. **Dispatch, dispatcher**

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

3. **Scheduling, scheduler**

   The processing to determine which task to execute next is called “scheduling” (or task scheduling).

   The kernel mechanism by which scheduling is realized is called a “scheduler” (or task scheduler).

Generally a scheduler is implemented within system call processing or in the dispatcher.

4. **Context**

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

5. **Precedence**

   The relationship among different 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.

2.2 Task States and Scheduling Rules

2.2.1 Task States

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

(a) RUN state

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

(b) READY state

The task has completed preparations for running, but cannot run because a task with higher precedence is running. In this state, the task is able to run whenever it becomes the task with the highest precedence among the tasks in READY or RUN state.

(c) Wait states

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

(c.1) WAIT state

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

(c.2) SUSPEND state

Execution was forcibly interrupted by another task.

(c.3) WAIT-SUSPEND state

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

(d) DORMANT state

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

(e) NON-EXISTENT state

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

Depending on the implementation, there may also be transient states that do not fall into any of the above categories (see section 2.4 on page 10).

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

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

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

Task state transitions in a typical implementation are shown in Figure 2.1. Depending on the implementation, there may be other states besides those shown here. A feature of µT-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). 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 RUN state and those from other states.

[Additional Notes]

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

Rationale for the Specification

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

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

Table 2.1: State Transitions Distinguishing Invoking Task and Other Tasks

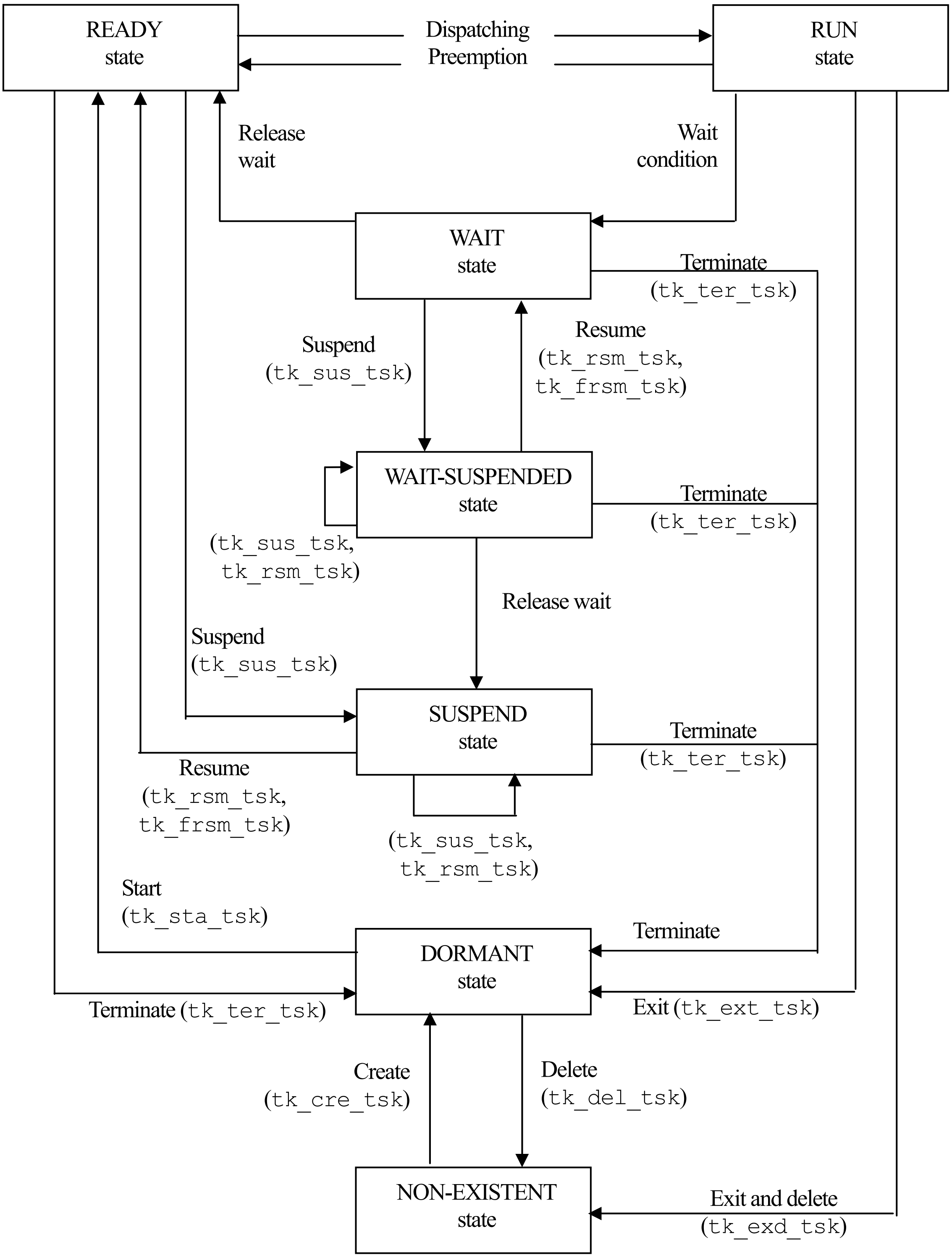


Figure 2.1: Task State Transitions

2.2.2 Task Scheduling Rules

The µT-Kernel specification 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 task with the highest precedence goes to RUN state and the others go to READY state. In determining precedence among tasks, of those tasks having different priority levels, the task with the highest priority has the highest precedence. Among tasks having the same priority, the one first going to a run state (RUN state or READY state) has the highest precedence. It is possible, however, to use a system call to change the precedence 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 RUN state is switched over. If no dispatch occurs, however, the switching of the task in RUN state is held off until dispatch occurs.

[Additional Notes]

According to the scheduling rules adopted in the µT-Kernel specification, so long as there is a high-precedence task in a run state, a task with lower precedence will simply not run. That is, unless the highest-precedence task goes to WAIT state or for some 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.

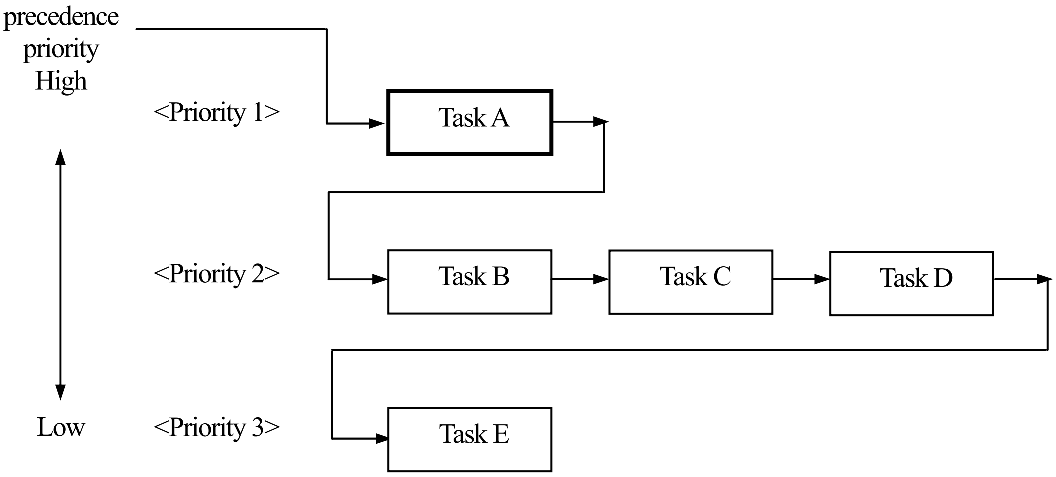


Figure 2.2: Precedence in Initial State

Figure 2.2 and following illustrate how the task that first goes to a run state (RUN state or READY state) gains precedence among tasks having the same priority. Figure 2.2 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 RUN state.

When Task A exits, Task B with the next-highest precedence goes to RUN state (Figure 2.3). 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. Therefore, the task precedence reverts to that in Figure 2.2.

Next, consider what happens when Task B goes to WAIT state in the conditions in Figure 2.3. Since task precedence is defined among tasks that can be run, the precedence among tasks becomes as shown in Figure

2.4. Thereafter, when the Task B wait state is released, Task B went to run state after Task C and Task D, and thus assumes the lowest precedence among tasks of the same priority (Figure 2.5). Summarizing the above, immediately after a task that went from READY state to RUN state reverts to READY state, it has the highest precedence among tasks of the same priority; but after a task goes from RUN state to WAIT state and then the wait is released, its precedence is the lowest among tasks of the same priority.

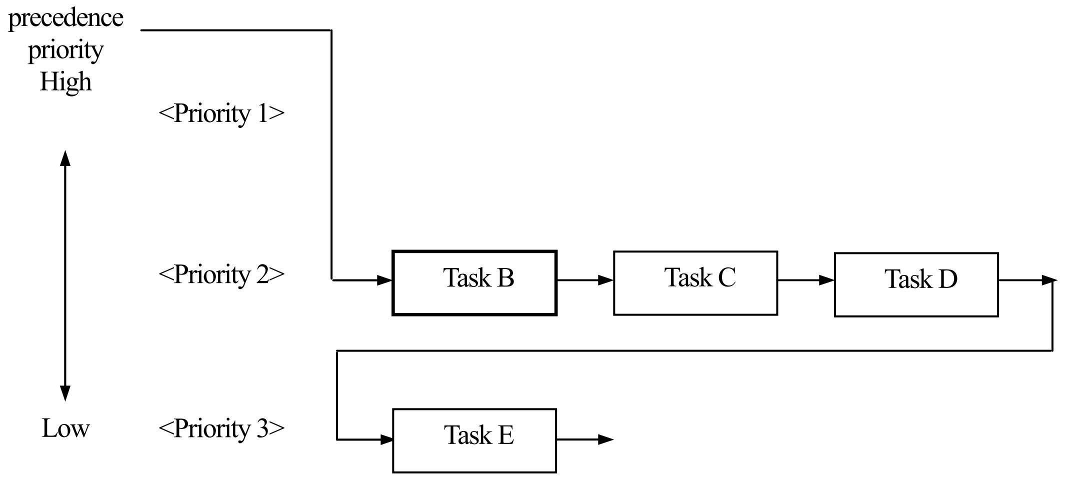


Figure 2.3: Precedence After Task B Goes To RUN State

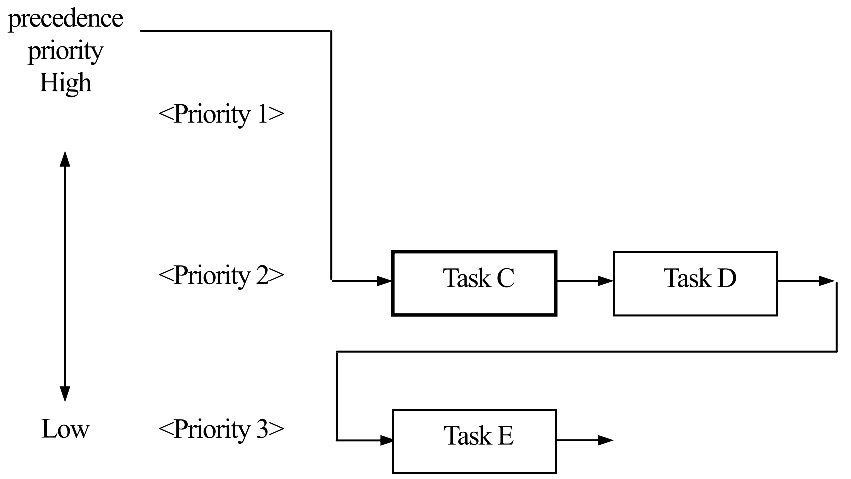


Figure 2.4: Precedence After Task B Goes To WAIT State

Note that after a task goes from SUSPEND state to a run state, it has the lowest precedence among tasks of the same priority.

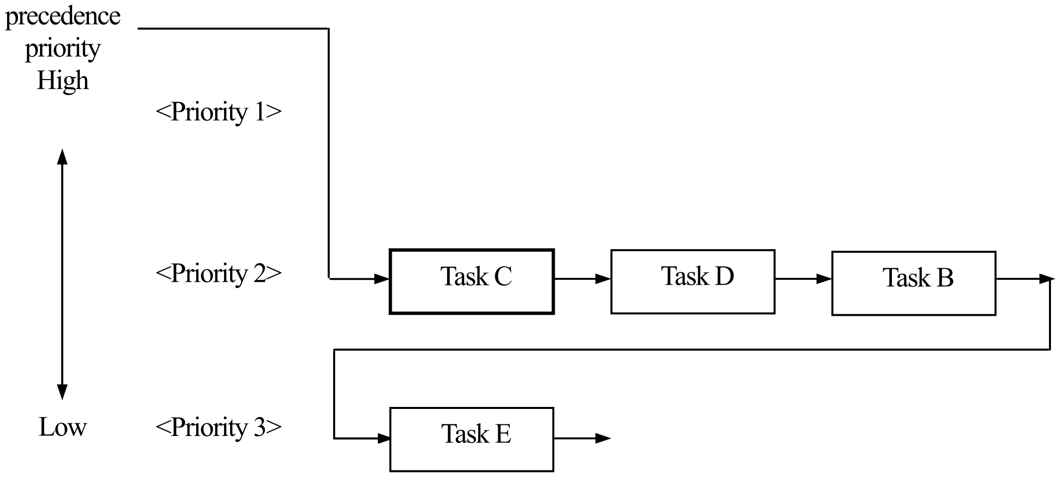


Figure 2.5: Precedence After Task B WAIT State Is Released

2.3 Interrupt Handling

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

2.4 System States

2.4.1 System States While Non-task Portion Is Executing

When programming tasks to run on μT-Kernel, the changes in task states can be tracked by looking at a task state transition diagram. In the case of routines such as interrupt handlers or extended SVC handlers, however, the user must perform programming at a level closer to the kernel than tasks. In this case, consideration must be made also of system states while a nontask portion is executing, otherwise programming cannot be done properly. An explanation of μT-Kernel system states is therefore given here. System states are classified as in Figure 2.6. Of these, a "transient state" is equivalent to OS running state (system call execution). From the standpoint of the user, it is important that each of the system calls issued by the user be executed indivisibly, and that the internal states while a system call is executing cannot be seen by the user. For this reason, the state while the OS is running is considered a "transient state" and internally it is treated as a blackbox. 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

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 user programs. These system calls should be used only in a subsystem or debugger that is so close to being an OS that it can be thought of as part of the OS. A task-independent portion and quasi-task portion are states while a handler is executing. The part of a handler that runs in a task context is a quasi-task portion, and the part with a context independent of a task is a task-independent portion. An extended SVC handler, which processes extended system calls defined by the user, is a quasi-task portion, whereas an interrupt handler or time event handler triggered by an external interrupt is a task-independent portion. In a quasi-task portion, tasks have the same kinds of state transitions as ordinary tasks, so system calls that enter WAIT state can be issued. A transient state, task-independent portion, and quasi-task portion are together called a nontask portion. When ordinary task programs are running, outside of these, this is "task portion running" state.

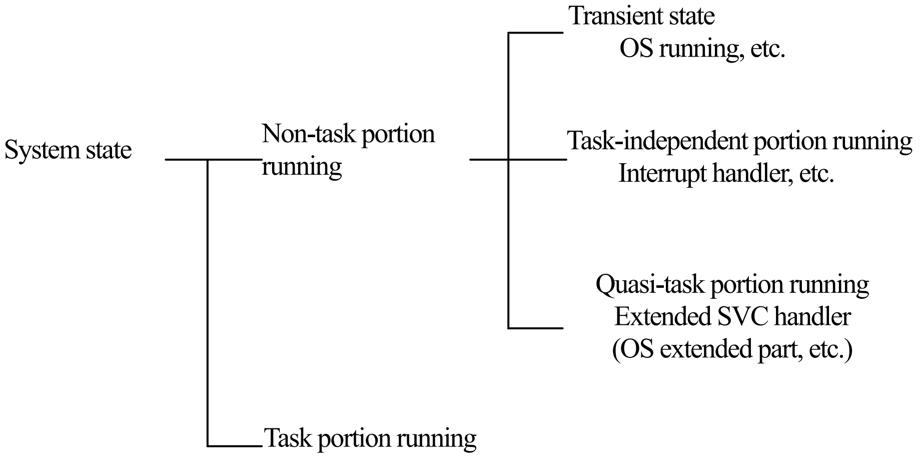


Figure 2.6: Classification of System States

2.4.2 Task-Independent Portion and Quasi-Task Portion

A feature of a task-independent portion (interrupt handlers, time event handlers, etc.) is that 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 WAIT 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 left over for execution after the task has been started by the dispatching, causing problems in case of interrupt nesting. This is illustrated in Figure 2.7.

In the example shown, 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 (1), starting Task B, the processing of parts (2) to (3) of Interrupt A would be put off until after Task B, with parts (2) to (3) executed only after Task A goes to RUN 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.

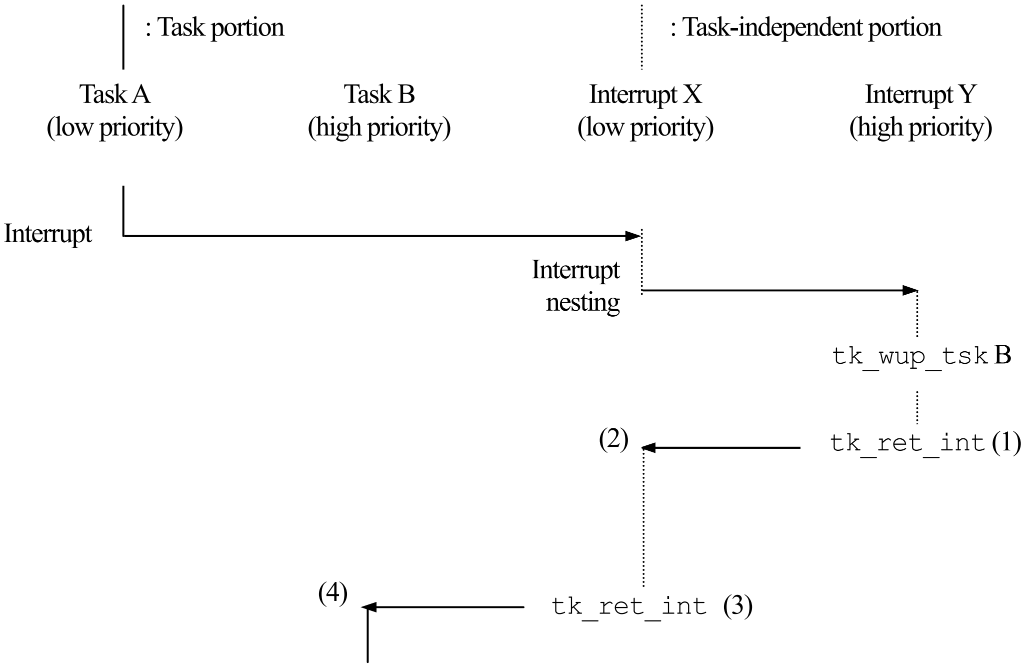
A feature of a quasi-task portion, on the other hand, is that the task executing prior to entering the quasi-task portion (the requesting task) can be identified, making it possible to define task states just as in the task portion; moreover, it is possible to enter WAIT state while in a quasi-task portion. 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 nontask 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.

- 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 parts 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 system call 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 RUN state. Task B is therefore executed before the rest of the extended SVC handler is completed. The rest of the extended SVC handler is executed after dispatching occurs again and Task A goes to RUN state.



If dispatching does not take place at (1), the remainder of the handler routine for Interrupt X ((2) to (3)) ends up being put off until later.

Figure 2.7: Interrupt Nesting and Delayed Dispatching

2.5 Objects

"Object" is the general term for resources handled by µT-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 differences in object behavior or the initial object state. When TA_XXXXX is specified for an object, that object is called a "TA_XXXXX attribute object" . If there is no particular attribute to be defined, TA_NULL (=0) is specified. Generally, there is no interface provided for reading attributes after an object is registered. In an object or handler 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. 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 and is passed in the parameters when the object starts execution. Extended information has no effect on µT-Kernel behavior and can be read by calling a system call to refer the status of the object. Each object is identified by an ID number. In the µT-Kernel, the ID number cannot be specified by users and is automatically allocated when an object is created. Therefore, it is difficult to identify objects when debugging. For this reason, when creating objects, users can specify an object name for debugging. This name is just for debugging and can be referenced only from debugger support functions. The µT-Kernel itself never checks the name of an object.

2.6 Memory

2.6.1 Address Space

µT-Kernel has only a single address space. That is, all tasks or task-independent portions always access the same address space.

[Difference with T-Kernel]

T-Kernel address space is partitioned into shared space and user space. The shared space can be accessed by all tasks likewise, and the user space can be accessed only by those tasks which belong to it.

User space does not exist in µT-Kernel, and only the equivalent of T-Kernel shared space exists in it.

2.6.2 Nonresident Memory

Virtual memory is not supported in µT-Kernel. Therefore, there are no such concepts as resident memory/ nonresident memory.

[Difference with T-Kernel]

Resident and nonresident memory exist in T-Kernel memory. The resident memory is always placed in actual memory, but the nonresident memory may be placed on disk, etc. based on the virtual memory mechanism.

Only the equivalent of T-Kernel resident memory exists in µT-Kernel.

2.6.3 Protection Levels

In T-Kernel, four stages of protection levels, from 0 to 3, can be specified. However, in µT-Kernel, systems without MMU are assumed, so each of the specified protection level shall be handled as protection level 0.

[Difference with T-Kernel]

The specification of handling protection levels exists in T-Kernel, and protection level 0 is assigned to OS or subsystems, protection level 1 is assigned to system applications, protection level 2 is reserved, and protection level 3 is assigned to user applications.

Chapter 3

Common µT-Kernel Specifications

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  unsigned char   UB;  /* unsigned 8-bit integer */
typedef  unsigned short  UH;  /* unsigned 16-bit integer */
typedef  unsigned long   UW;  /* unsigned 32-bit integer */

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

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

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

typedef  INT             ID;  /* general ID */
typedef  W             MSEC;  /* general time (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 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 code */
#define TNULL        ((TC)0)  /* TRON code string termination */
```

[Additional Notes]

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

Parameters such as stksz, wupcnt, and message size that clearly do not take negative values are also in principle signed integer (INT, W) data type. This is in keeping with the overall TRON rule that integers should be treated as signed numbers to the extent 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.).

3.1.2 Other Defined Data Types

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

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

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

/*
*  Common constants
*/

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

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

[Difference with T-Kernel 1.00.00]

The 'signed' is explicitly specified in the signed type definition. Also, 32-bit type definition is defined by 'long' instead of 'int'.

W is used instead of INT and UW instead of UINT in some type definitions. This is because of the idea that the types with 32-bit width should be W or UW.

3.2 System Calls

3.2.1 System Call Format

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

The method for interfacing at the assembly language level is implementation-dependent and not standardized here. Calling by means of a C language interface is recommended even when an assembly language program is created. In this way, portability is assured for programs written in assembly language even if the OS changes, as long as the CPU is the same.

The following common rules are established for system call interfaces.

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

The implementation method of system call interfaces shall be implementation-defined. For example, implementations using C language's macro, inline function, and inline assembler, etc. are possible.

[Difference with T-Kernel]

In T-Kernel, system call interfaces are specified to be as a library, but in μT-Kernel, it is implementation-defined. This is because, in μT-Kernel, performance efficiency takes precedence over portability among different compilers.

3.2.2 System Calls Possible from Task-Independent Portion and Dispatching Prohibited State

The following system calls should run even if they are issued from task-independent portion and dispatch disabled state.

tk_sta_tsk	Start task
tk_wup_tsk	Wakeup task
tk_rel_wai	Release wait
tk_sus_tsk	Suspend task
tk_sig_sem	Signal semaphore
tk_set_flg	Set event flag
tk_rot_rdq	Rotate task queue
tk_get_tid	Get task ID
tk_sta_cyc	Start cyclic handler
tk_stp_cyc	Stop cyclic handler

`tk_sta_alm`	Start alarm handler
`tk_stp_alm`	Stop alarm handler
`tk_ref_tsk`	Reference task status
`tk_ref_cyc`	Reference cyclic handler status
`tk_ref_alm`	Reference alarm handler status
`tk_ref_sys`	Reference system status
`tk_ret_int`	Return from interrupt handler

Also, the following system calls should run in the same way as those issued from dispatch enabled state (also, never return "E_CTX") even if they are issued from dispatch disabled state.

`tk_fwd_por`	Forward rendezvous to other port
`tk_rpl_rdv`	Reply rendezvous

If system calls besides these are issued from task-independent portion and dispatch disabled state, the behavior shall be implementation-defined.

[Difference with T-Kernel]

`tk_sig_tev` does not exist in the list of system calls which are allowed to be executed from task-independent portion and dispatch disabled state. This is because this system call does not exist in µT-Kernel.

3.2.3 Restricting System Call Invocation

In µT-Kernel, systems without MMU are assumed and the protection level functions are not supported (everything is handled as a protection level 0). Therefore, there are no restrictions concerning invocations of system calls by protection level.

3.2.4 Modifying a Parameter Packet

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

Additional information that is implementation-dependent may be added to a parameter packet. However, changing the data types and the order of information defined in the standard specification or deleting any of this information is not allowed. When implementation-dependent information is added, it must be placed after the standard defined information. When implementation-dependent information is added to a packet of input information passed to a system call (T_CTSK, etc.), if the system call is invoked while this additional information is not yet initialized (memory contents indeterminate), the system call must still function normally.

Ordinarily a flag indicating that valid values are set in the additional information is defined in the attribute flag implementation-dependent area included in the standard specification. When that flag is set (1), the additional information is to be used; and when the flag is not set (0), the additional information is not initialized (memory contents indeterminate) and the default settings are to be used. The reason for this specification is to ensure that a program developed within the scope of the standard specification will be 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 actual values for function codes of system calls are implementation-defined, but unique negative values shall be assigned to each system call. The reason why function codes are implementation-defined is that system call interfaces are implementation-defined in µT-Kernel and implementations called in a form that does not use function codes may also exist.

Positive values are assigned to the function codes of extended SVCs. For details, refer to `tk_def_ssy`.

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 separately for each system call. An exception to this principle is that there are some system calls that do not return when called. A system call that does not return is declared in the C language API as having no return code (that is, a void type function).

The error code is classified into error classes based on the necessity of detection or the occurrence. For error class classification, refer to "5.2 Error Code List".

[Difference with T-Kernel]

In T-Kernel, the lower 16 bits are set to sub error code and upper 16 bits are set to main error code, but only main error code exists in µT-Kernel. For example, the main error code of E_ID is defined as -18, but in T-Kernel, the upper 16 bits are set to -18 and the lower 16 bits are set to zero. On the other hand, in µT-Kernel, E_ID is defined as -18. To ensure compatibility, you should use macros such as E_ID instead of actual values as error codes.

While macros for processing error codes such as ERCD, MERCD and SERCD exist in T-Kernel, these macros do not exist in µT-Kernel. When porting an application that requires those macros to µT-Kernel, it is necessary to modify the application not to use those macros or to define those macros with the same names as in T-Kernel.

3.2.7 Timeout

A system call that may enter WAIT state has a timeout function. If processing is not completed by the time the specified timeout interval has elapsed, the processing is canceled and the system call returns 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 WAIT state even when a situation arises in which it would ordinarily go to WAIT state. In other words, a system call with timeout set to 0 when it is invoked has no possibility of entering WAIT state. Invoking a system call with timeout set to 0 is called polling; that is, a system call that performs polling has no chance of entering WAIT state.

The descriptions of individual system calls, as a rule, describe the behavior when there is no timeout (in other words, when an endless wait occurs). Even if the system call description says that the system call “enters WAIT state” or “is put in WAIT state”, if a timeout is set and that time interval elapses before processing is completed, the WAIT state is released and the system call returns error code `E_TMOUT`. In the case of polling, the system call returns `E_TMOUT` without entering WAIT state. Timeout (TMO type) may be a positive integer, `TMO_POL` (= 0) for polling, or `TMO_FEVR` (= -1) for endless wait. If a timeout interval is set, the timeout processing must be guaranteed to take place even after the specified interval from the system call issuing has elapsed.

[Additional Notes]

Since a system call that performs polling does not enter WAIT state, there is no change in the precedence of the task calling it. In a general implementation, when the timeout is set to 1, timeout processing takes place on the second time tick after a system call is invoked. Since a timeout of 0 cannot be specified (0 being allocated to `TMO_POL`), in this kind of implementation timeout does not occur on the initial time tick after the system call is invoked.

3.2.8 Relative Time and System Time

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

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

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

```
typedef UW RELTIM;
```

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

```
typedef W TMO;
```

  Endless 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 or TMO has elapsed. For example, if the timer interrupt cycle is 1 ms and a timeout of 1 ms is specified, timeout occurs on the second timer interrupt (because the first timer interrupt does not exceed 1 ms).

[Difference with T-Kernel 1.00.00]

The RELTIM type is defined as UW instead of UINT. Also, the TMO type is defined by W type instead of INT type. This is because the INT/UINT type was assumed to have insufficient range to represent the time in 16-bit environments.

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 a 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 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 OS, the task start address or handler address is a parameter pointing 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 (return or "}").

However, in the case of tasks, even if a high-level language support routine is used, tasks should not end by a return from function. In the case of a task exception handler, the high-level language support routine is supplied as source code and is to be embedded in the user program.

The internal working of a high-level language support routine is as illustrated in Figure 3.1.

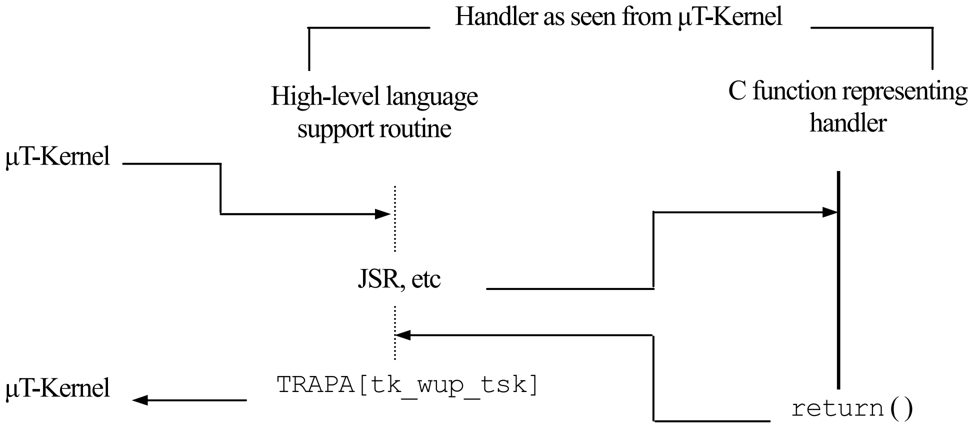

Handler as seen from μT-Kernel
High-level language support routine
C function representing handler
μT-Kernel
JSR, etc
μT-Kernel
TRAPA[tk_wup_tsk]
return()

Figure 3.1: Behavior of High-Level Language Support Routine

[Difference with T-Kernel]

Therefore, to ensure compatibility with T-Kernel, the task should not end by a return from function. In T-Kernel, the concept of protection level exists. However, in systems with MMU, it is difficult to achieve high-level language support routines in the tasks or task exception handlers working at a protection level different from the OS, although it is possible to achieve high-level language support routines with relative ease in interrupt handlers, etc. working at the same protection level as the OS. So, in T-Kernel, task ending by a return from function is supposed to be not guaranteed. In μT-Kernel, MMU is not a precondition and no protection level exists, so the implementation may allow you to end a task by a return from function. However, taking compatibility with T-Kernel into account, you should not use this feature even if it is possible.

Chapter 4

µT-Kernel Functions

This chapter describes the system calls provided by the µT-Kernel in detail.

µT-Kernel has the following features:

- task management
- task-dependent synchronization
- synchronization communication
- extended synchronization communication
- memory pool management
- time management
- interrupt management
- system state management
- subsystem management
- device management
- interrupt management
- debugger support

It depends on the OS provider's judgment whether or not to implement the debugger support functions and the debugger support related functions in each function because they are functions for debugging. However, when implemented, the interface specified in the specification should be observed. Also, when providing other functions concerning debugging, their names should be different from those specified in the specification.

[Difference with T-Kernel]

Since no protection levels exist in µT-Kernel, there are no classifications such as T-Kernel/OS and T-Kernel/SM. Also, classifications such as T-Kernel/DS do not exist and the equivalent features are set as debugger support functions.

In µT-Kernel, the following features are not present:

- task exception support
- system memory management
- address space management
- I/O port access support
- power saving
- system configuration information management

Also, among the following functions, some system calls do not exist:

- task management

- task-dependent synchronization
- system state management
- subsystem management
- interrupt management

The description of each function shows you which system calls exist or not.

On the other hand, the function present in µT-Kernel but not in T-Kernel is the ability to specify a memory area to be used by the system call. This function specifies the area allocated by the user as a memory area to be used for task stack or message buffer memory pool. This function allows adaptations such as allocation of the stack to high-speed internal memory. If this function is not used, the OS allocates the memory area.

For the specifics of the usage of this function, refer to the descriptions of `tk_cre_tsk`, `tk_cre_mbf`, `tk_cre_mpf`, and `tk_cre_mpl`.

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 starting and exiting a task, canceling a task start request, 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.

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

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

[Difference with T-Kernel]

The following system calls do not exist:

- `tk_chg_slt` modifies a task slice time
- `tk_get_tsp` gets the user space
- `tk_set_tsp` sets the user space
- `tk_get_rid` retrieves a resource group belonging to the task
- `tk_set_rid` sets a resource group belonging to the task
- `tk_get_cpr` retrieves the coprocessor register
- `tk_set_cpr` sets the coprocessor register
- `tk_inf_tsk` refers to task statistical data

tk_cre_tsk

Create Task

[C Language Interface]

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

[Parameters]

T_CTSK* pk_ctsk Information about the task to be created

pk_ctsk detail:

VP	exinf	Extended information
ATR	tskatr	Task attributes
FP	task	Task start address
PRI	itskpri	Initial task priority
W	stksz	Stack size (bytes)
UB	dsname[8]	DS object name
VP	bufptr	User buffer pointer

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

[Return Parameters]

ID	tskid	Task ID
	or	Error Code

[Error Codes]

E_NOMEM	Insufficient memory (memory for control block or user stack cannot be allocated)
E_LIMIT	Number of tasks exceeds the system limit
E_RSATR	Reserved attribute (tskatr is invalid or cannot be used), or the specified co-processor does not exist
E_PAR	Parameter Error

[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 the DORMANT state.

itskpri specifies the initial priority at the time the task is started.

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

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

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

```
tskatr     :=(TA_ASM||TA_HLNG)
      |[TA_USERBUF]|[TA_DSNAME]|(TA_RNG0||TA_RNG1||TA_RNG2||TA_RNG3)
```

```
TA_ASM        Indicates that the task is written in assembly language
TA_HLNG       Indicates that the task is written in high-level language
TA_USERBUF    Indicates that the task uses an area specified by user as stack
TA_DSNAME     Specifies DS object name
TA_RNGn       Indicates that the task runs at protection level n
```

The ability to specify 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   /* Assembly  program             */
#define  TA_HLNG       0x00000001   /* High-level language program   */
#define  TA_USERBUF    0x00000020   /* User buffer                   */
#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     */
```

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

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

The task cannot (must not) be terminated by a simple return from the function.

The form of the task when the `TA_ASM` attribute is specified is implementation-dependent, but `stacd` and `exinf` must be passed as startup parameters.

If `TA_USERBUF` is specified, `bufptr` becomes valid, and the memory area of `stksz` bytes with `bufptr` at the head is used as a stack area. In this case, the stack is not prepared by OS. If TA_USERBUF is not specified, `bufptr` is ignored, and a stack area is allocated by the OS.

Each task has a stack area. There is no partition between system stack and user stack like T-Kernel because each protection level is always handled as protection level 0 in µT-Kernel.

When `TA_DSNAME` is specified, `dsname` is valid and specifies the DS object name. DS object name is used by the debugger to identify objects and is handled only by the debugger support functions API, td_ref_dsname and td_set_dsname. For more details, refer to 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 always runs as protection level 0 because functions concerning protection level are not supported in µT-Kernel.

In a system with a separate interrupt stack, interrupt handlers also use the system stack. An interrupt handler runs at protection level 0.

The 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.

[Difference with T-Kernel]

There is no system stack size (sstksz), user space page table (uatb), logical space ID (lsid), or resource ID (resid) in pk_ctsk. TA_TASKSPACE, TA_RESID, and TA_COPn are not specified as attributes allowable for tskatr. TA_USERBUF and bufptr have been added.

In µT-Kernel, a system without MMU is assumed, system stack and user stack are automatically bifurcated by the system, and the number of stacks is supposed to be only one. Consequently, there is no specification concerning system stack. For the same reason, functions concerning user space are omitted.

Also, subsystem functions are focused only on extended SVCs, so functions concerning resource IDs are omitted.

There is no coprocessor specification because in small systems targeted by µT-Kernel, the function to generically handle the coprocessor is deemed to be unnecessary.

In µT-Kernel, systems without MMU are assumed, so functions concerning protection level are not supported. Nevertheless, TA_RNGn attributes have been retained to maintain compatibility with T-Kernel.

[Difference with T-Kernel 1.00.00]

The type of stksz (stack size), the member of T_CTSK, is W instead of INT.

[Porting Guideline]

TA_USERBUF and bufptr are not present in T-Kernel. So, if this ability is used , modifications are needed when porting to T-Kernel. However, if stksz is set correctly, you can port it by simply deleting TA_USERBUF and bufptr.

tk_del_tsk

Delete Task

[C Language Interface]

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

[Parameters]

ID	tskid	Task ID

[Return Parameters]

ER	ercd	Error code

[Error Codes]

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

[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.

tk_sta_tsk

Start Task

[C Language Interface]

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

[Parameters]

ID	tskid	Task ID
INT	stacd	Task start code

[Return Parameters]

ER ercd	Error code

[Error Codes]

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

[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.

[Porting Guideline]

Note that stacd is of INT type and the allowable range of values that can be specified may vary depending on the system.

tk_ext_tsk

Exit Task

[C Language Interface]

```
void tk_ext_tsk ( ) ;
```

[Parameters]

None

[Return Parameters]

Does not return to the context issuing the system call.

[Error Codes]

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

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

[Description]

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

[Additional Notes]

When a task terminates by `tk_ext_tsk`, the resources acquired by the task up to that time (memory blocks, semaphores, etc.) are not automatically released. 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 hang. For this reason, these system calls do not return even if an error is detected. As a rule, the task priority and other information included in the TCB is reset when the task returns to DORMANT state. If, for example, the task priority is changed by `tk_chg_pri` and later terminated by `tk_ext_tsk`, the task priority reverts to the startup priority (`itskpri`) specified when the task was started. It does not keep the task priority in effect at the time `tk_ext_tsk` was executed. System calls that do not return to the calling context are those named `tk_ret_???` or `tk_ext_???` (`tk_exd_???`).

tk_exd_tsk

Exit and Delete Task

[C Language Interface]

```
void tk_exd_tsk ( ) ;
```

[Parameters]

None.

[Return Parameters]

Does not return to the context issuing the system call.

[Error Codes]

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

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

[Description]

Terminates the invoking task normally and also deletes it.

This system call changes the state of the invoking task to NON-EXISTENT state (no longer exists in the system).

[Additional Notes]

When a task terminates by `tk_exd_tsk`, the resources acquired by the task up to that time (memory blocks, semaphores, etc.) are not automatically released. 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 hang. For this reason, these system calls do not return even if an error is detected.

tk_ter_tsk

Terminate Task

[C Language Interface]

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

[Parameters]

```
ID tskid Task ID
```

[Return Parameters]

```
ER          ercd        Error code
```

[Error Codes]

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

[Description]

Forcibly terminates the task specified in tskid.

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

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

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

The relationships between target task states and the results of executing tk_ter_tsk are summarized in Table 4.1.

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

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 released. The user is responsible for releasing such resources before the task is terminated.

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

Forcible termination of another task is intended for use only by a debugger or a few other tasks closely 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 irrespective of the running state of the target task.

If, for example, a task is forcibly terminated while the task is calling a middleware function, the task would terminate right in the middle of the middleware execution. If such a situation is allowed, normal operation of the middleware cannot be guaranteed. This is an example of how task termination cannot be allowed when the task status (what it is being executed) is unknown. Ordinary applications therefore must not use the forcible termination function.

tk_chg_pri

Change Task Priority

[C Language Interface]

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

[Parameters]

ID	tskid	Task ID
PRI	tskpri	Task priority

[Return Parameters]

ER	ercd	Error code

[Error Codes]

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

[Description]

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

Task priority values are specified from 1 to 140 with the smaller numbers indicating higher priority. When TSK_SELF (= 0) is specified in tskid, the invoking task is the target task. Note, however, that when tskid = TSK_SELF is specified in a system call issued from a task-independent portion, error code E_ID is returned. When TPRI_INI (= 0) is specified as tskpri, the target task base priority is changed to the initial priority when the task was started (itskpri).

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

If, as a result of this system call execution, the target task current priority matches the base priority (this condition is always met when the mutex function is not used), processing is as follows. If the target task is in a run state, the task precedence changes according to its priority. The target task has the lowest precedence among tasks of the same priority after the change. If the target task is in some kind of priority-based queue, the order in that queue changes in accordance with the new task priority. Among tasks of the same priority after the change, the target task is queued at the end.

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

[Additional Notes]

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

In some cases, when this system call results in a base priority change while the target task is waiting for a `TA_INHERIT` attribute mutex lock, dynamic priority inheritance processing may be necessary. When a mutex function is not used and the system call is issued specifying the invoking task as the target task, setting the new priority to the base priority of the invoking task, the order of execution of the invoking task becomes the lowest among tasks of the same priority. This system call can therefore be used to relinquish execution privilege.

tk_get_reg

Get Task Registers

[C Language Interface]

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

[Parameters]

ID	tskid	Task ID

[Return Parameters]

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

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

[Error Codes]

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

[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 OS as virtual registers.

tk_set_reg

Set Task Registers

[C Language Interface]

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

[Parameters]

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

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

[Return Parameters]

ER	ercd	Error code

[Error Codes]

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

[Description]

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

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

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

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

tk_ref_tsk

Reference Task Status

[C Language Interface]

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

[Parameters]

ID	tskid	Task ID
T_RTSK*	pk_rtsk	Address of packet for returning task status

[Return Parameters]

ER	ercd	Error code

pk_rtsk detail:

VP	exinf	Extended information
PRI	tskpri	Current task priority
PRI	tskbpri	Base priority
UINT	tskstat	Task state
UW	tskwait	Wait factor
ID	wid	Waiting object ID
INT	wupcnt	Queued wakeup requests
INT	suscnt	Nested suspend requests

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

[Error Codes]

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

[Description]

Gets the state of the task specified in tskid.

tskstat takes the following values.

tskstat:	TTS_RUN	0x0001	RUN
	TTS_RDY	0x0002	READY
	TTS_WAI	0x0004	WAIT
	TTS_SUS	0x0008	SUSPEND
	TTS_WAS	0x000c	WAIT-SUSPEND
	TTS_DMT	0x0010	DORMANT

```
TTS_NODISWAI     0x0080       Wait state disabled
```

Task states such as TTS_RUN and TTS_WAI are expressed by corresponding bits, which is useful when making a complex state decision (e.g., deciding that the state is one of either RUN or READY state).

Note that, of the above states, TTS_WAS is a combination of TTS_SUS and TTS_WAI, but TTS_SUS is never combined with other states (TTS_RUN, TTS_RDY, and TS_DMT). In the case of TTS_WAI (including TTS_WAS), if wait states are disabled by `tk_dis_wai`, TTS_NODISWAI is set. TTS_NODISWAI is never combined with states other than TTS_WAI.

When `tk_ref_tsk` is executed for an interrupted task from an interrupt handler, RUN (TTS_RUN) is returned as `tskstat`.

When `tskstat` is TTS_WAI (including TTS_WAS), the values of `tskwait` and `wid` are as shown in Table 4.2.

tskwait	Value	Description	wid
TTW_SLP	0x00000001	Wait caused by `tk_slp_tsk`	0
TTW_DLY	0x00000002	Wait caused by `tk_dly_tsk`	0
TTW_SEM	0x00000004	Wait caused by `tk_wai_sem`	semid
TTW_FLG	0x00000008	Wait caused by `tk_wai_flg`	flgid
TTW_MBX	0x00000040	Wait caused by `tk_rcv_mbx`	mbxid
TTW_MTX	0x00000080	Wait caused by `tk_loc_mtx`	mtxid
TTW_SMBF	0x00000100	Wait caused by `tk_snd_mbf`	mbfid
TTW_RMBF	0x00000200	Wait caused by `tk_rcv_mbf`	mbfid
TTW_CAL	0x00000400	Wait caused by `tk_rcv_mbf`	porid
TTW_ACP	0x00000800	Wait for rendezvous acceptance	porid
TTW_RDV	0x00001000	Wait for rendezvous completion	0
(TTW_CAL|TTW_RDV)	0x00001400	Wait on rendezvous call or wait for rendezvous completion	0
TTW_MPF	0x00002000	Wait for `tk_get_mpf`	mpfid
TTW_MPL	0x00004000	Wait for `tk_get_mpl`	mplid

Table 4.2: Values of tskwait and wid

When `tskstat` is not TTS WAI (including TTS WAS), both `tskwait` and `wid` are 0.

For a task in DORMANT state, `wupcnt` = 0 and `suscnt` = 0.

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

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

[Additional Notes]

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

[Difference with T-Kernel]

Maximum consecutive execution time (slicetime), wait-disabled waiting factor (waitmask), permitted task exception (texmask), and occurred task event (tskevent) are deleted from T_RTSK. Non-disable wait state (TTS_NODISWAI) is deleted from the values set to tskstat, and the value concerning task event (TTW_EVn) is deleted from the values set to tskwait. This is because, in µT-Kernel, there are no functions concerning timesharing execution, wait-disabled state, task exception, and task event.

[Difference with T-Kernel 1.00.00]

tskwait, the member of T_RTSK, is of UW type instead of UINT type. In µT-Kernel, 16-bit width is sufficient for this member, but 32-bit width is required for T-Kernel. Therefore, this member is modified to UW type based on the policy that those that require 32-bit width are set to the W/UW types whose bit widths are explicitly defined.

4.2 Task-Dependent Synchronization Functions

Task-dependent synchronization functions achieve synchronization among tasks by direct manipulation of task states. They include functions for task sleep and wakeup, for canceling wakeup requests, for forcibly releasing task WAIT state, for changing a task state to SUSPEND state, for delaying execution of the invoking task, and for disabling task WAIT 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. Queuing of task wakeup requests is realized by having the task keep a task wakeup request queuing count. When the task is started, this count is cleared to 0.

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

[Difference with T-Kernel]

There are no system calls concerning the following task events and wait-disabled state:

- `tk_sig_tev` sends a task event
- `tk_wai_tev` waits for a task event
- `tk_dis_wai` disables task wait state
- `tk_ena_wai` clears task wait-disabled

tk_slp_tsk

Sleep Task

[C Language Interface]

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

[Parameters]

TMO	tmout	Timeout

[Return Parameters]

ER	ercd	Error code

[Error Codes]

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

[Description]

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

If tk_wup_tsk is issued for the invoking task before the time specified in tmout has elapsed, this system call completes normally. If timeout occurs before tk_wup_tsk is issued, error code E_TMOUT is returned.

Specifying tmout = TMO_FEVR =(-1) means endless 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 a wait state, tk_slp_tsk can never be nested. It is possible, however, for another task to issue tk_sus_tsk for a task that was put in a wait state by tk_slp_tsk. In this case, the task goes to WAIT-SUSPEND state.

For simply delaying a task, tk_dly_tsk should be used rather than tk_slp_tsk. The task sleep function is intended for use by applications and as a rule should not be used by middleware. The reason is that attempting to achieve synchronization by putting a task to sleep in two or more places would cause confusion, leading to mis-operation. For example, if sleep were used by both an application and middleware for synchronization, a wakeup request might arise in the application while middleware has the task sleeping. In such a situation, normal operation would not be possible in either the application or middleware. Proper task synchronization is not possible because it is not clear where the wait for wakeup originated. Task sleep is often used as a simple means of task synchronization. Applications should be able to use it freely, which means, as a rule, it should not be used by middleware.

[Difference with T-Kernel]

`E_DISWAI` does not exist in error codes. This is because in µT-Kernel, there is no wait-disabled function.

tk_wup_tsk

Wakeup Task

[C Language Interface]

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

[Parameters]

ID	tskid	Task ID

[Return Parameters]

ER	ercd	Error code

[Error Codes]

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

[Description]

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

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

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

The queuing of wakeup requests works as follows:

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

It is always possible to queue tk_wup_tsk once (wupcnt=1). The maximum value of wake-up request queueing count (wupcnt) is implementation-defined and specifies an appropriate value greater than or equal to one. That is, an error does not occur if tk_wup_tsk is issued once to a task in non-wait state, however it is implementation-defined concerning whether second or subsequent issuances of tk_wup_tsk cause an error or not.

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

tk_can_wup

Cancel Wakeup Task

[C Language Interface]

```
INT wupcnt = tk_can_wup ( ID tskid ) ;
```

[Parameters]

ID	tskid	Task ID

[Return Parameters]

INT	wupcnt	Number of queued wakeup requests
	or	ErrorCode

[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)

[Description]

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

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

[Additional Notes]

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

tk_rel_wai

Release Wait

[C Language Interface]

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

[Parameters]

ID	tskid	Task ID

[Return Parameters]

ER	ercd	Error code

[Error Codes]

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

[Description]

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

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

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

The tk_rel_wai system call does not release a SUSPEND state. If it is issued for a task in WAITSUSPEND state, the task goes to SUSPEND state. If it is also necessary to release SUSPEND state, the system call tk_frsm_tsk is issued instead of this system call. The states of the target task when tk_rel_wai is called and the results of its execution in each state are shown in Table 4.3.

[Additional Notes]

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

The main differences between tk_rel_wai and tk_wup_tsk are the following:

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

Target Task State	tk_rel_wai ercd Parameter	Processing
Run state (RUN, READY) (not for invoking task)	E_OBJ	No operation
RUN state (for invoking task)	E_OBJ	No operation
WAIT state	E_OK	Wait released.*
SUSPEND state	E_OBJ	No operation
WAIT-SUSPEND state	E_OK	Transition to SUSPEND state
DORMANT state	E_OBJ	No operation
NON-EXISTENT state	E_NOEXS	No Operation

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

Table 4.3: Task States and Results of `tk_rel_wai` Execution

- From the perspective of the task in WAIT state, release of the WAIT state by `tk_wup_tsk` returns a Normal completion (`E_OK`), whereas release by `tk_rel_wai` returns an error code (`E_RLWAI`).
- Wakeup requests by `tk_wup_tsk` are queued if `tk_slp_tsk` has not yet been executed. If `tk_rel_wai` is issued for a task not in WAIT state, error code `E_OBJ` is returned.

tk_sus_tsk

Suspend Task

[C Language Interface]

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

[Parameters]

ID	tskid	Task ID

[Return Parameters]

ERercdError code

[Error Codes]

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

[Description]

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

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

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

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

When this system call is issued from a task-independent portion, if a task in RUN 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 SUSPEND state multiple times. This is called nesting of suspend requests. In this case, the task reverts to its original state only when tk_rsm_tsk has been issued for the same number of times as tk_sus_tsk (suscnt). Accordingly, nesting of the pair tk_sus_tsk and tk_rsm_tsk is possible.

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

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

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

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

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

tk_rsm_tsk
tk_frsm_tsk

Resume Task
Resume Task Force

[C Language Interface]

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

[Parameters]

ID	tskid	Task ID

[Return Parameters]

ER	ercd	Error code

[Error Codes]

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

[Description]

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

When the target task is in a combined WAIT state and SUSPEND state (WAIT-SUSPEND state), executing `tk_rsm_tsk` releases only the SUSPEND state, putting the task in WAIT state. This system call cannot be issued for the invoking task. If the invoking task is specified, error code `E_OBJ` is returned.

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

If the target task is neither suspended (SUSPEND state) nor WAITING-SUSPENDED (WAIT-SUSPEND state), error code E_OBJ is returned.

[Additional Notes]

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

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

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

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

tk_dly_tsk

Delay Task

[C Language Interface]

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

[Parameters]

RELTIM	dlytim	Delay time

[Return Parameters]

ER	ercd	Error code

[Error Codes]

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

[Description]

Temporarily stops execution of the invoking task and waits for time dlytim to elapse. The state while the task waits for the delay time to elapse is a WAIT state and is subject to release by tk_rel_wai.

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

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

[Additional Notes]

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

[Difference with T-Kernel]

E_DISWAI does not exist in error codes. This is because in µT-Kernel, there is no wait-disabled function.

4.3 Synchronization and Communication Functions

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

[Difference with T-Kernel]

In µT-Kernel, since there is no wait-disabled function, attribute `TA_NODISWAI` and error code `E_DISWAI` do not exist.

4.3.1 Semaphore

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

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

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

tk_cre_sem

Create Semaphore

[C Language Interface]

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

[Parameters]

T_CSEM*	pk_csem	Information about the semaphore to be created

pk_csem detail:

VP	exinf	Extended information
ATR	sematr	Semaphore attributes
INT	isemcnt	Initial semaphore count
INT	maxsem	Maximum semaphore count
UB	dsname[8]	DS object name

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

[Return Parameters]

ID	semid	Semaphore ID
	or	Error Code

[Error Codes]

E_NOMEM	Insufficient memory (memory for control block cannot be allocated)
E_LIMIT	Semaphore count exceeds the system limit
E_RSATR	Reserved attribute (sematr is invalid or cannot be used)
E_PAR	Parameter error (pk_csem is invalid; isemcnt or maxsem is negative or invalid)

[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. Note that the highest value that can be assigned to maxsem is implementation-defined, but shall be at least 32767.

exinf can be used freely by the user to store miscellaneous information about the created semaphore.

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

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

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

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

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

TA_FIRST and TA_CNT specify precedence of resource acquisition. TA_FIRST and TA_CNT do not change the order of the queue, which is determined by TA_TFIFO or TA_TPRI. When TA_FIRST is specified, resources are allocated starting from the first task in the queue regardless of request count. As long as the first task in the queue cannot obtain the requested number of resources, tasks behind it in the queue are prevented from obtaining resources.

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

When TA_DSNAME is specified, dsname is valid and specifies the DS object name. DS object name is used by the debugger to identify objects and is handled only by debugger support functions API, td_ref_dsname and td_set_dsname. For more details, refer to 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 is specified              */
```

[Difference with T-Kernel]

TA_NODISWAI does not exist in the attribute for sematr, and E_DISWAI does not exist in error codes. This is because there is no wait-disabled function in µT-Kernel.

[Difference with T-Kernel 1.00.00]

The value to be specified for maxsem is 32767 instead of 65535. This is because in a 16-bit environment, the INT type can store only up to 32767 in most cases.

tk_del_sem

Delete Semaphore

[C Language Interface]

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

[Parameters]

ID	semid	Semaphore ID

[Return Parameters]

ER	ercd	Error code

[Error Codes]

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

[Description]

Deletes the semaphore specified in semid.

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

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

tk_sig_sem

Signal Semaphore

[C Language Interface]

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

[Parameters]

ID	semid	Semaphore ID
INT	cnt	Resource return count

[Return Parameters]

ER	ercd	Error code

[Error Codes]

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

[Description]

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

In some conditions, more than one task may be allocated resources and put in the READY state. If the semaphore count increases to the point where the maximum count (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]

An error is not returned even if semcnt exceeds the semaphore initial count (isemcnt). When semaphores are used for synchronization (like tk_wup_tsk and tk_slp_tsk) and not for mutual exclusion control, 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 occurrence of an error when the count increases.

tk_wai_sem

Wait on Semaphore

[C Language Interface]

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

[Parameters]

ID	semid	Semaphore ID
INT	cnt	Resource request count
TMO	tmout	timeout

[Return Parameters]

ER	ercd	Error code

[Error Codes]

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

[Description]

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

A maximum wait time (timeout) can be set in tmout. If the tmout time elapses before the wait release condition is met (tk_sig_sem is not executed), the system call terminates returning timeout error code E_TMOUT. Only positive values can be set in tmout. The time base for tmout (time unit) is the same as that for system time (= 1 ms).

When TMO_POL = 0 is set in tmout, this means 0 was specified as the timeout value, and E_TMOUT is returned without entering WAIT state even if no resources are acquired.

When TMO_FEVR =(-1) is set in tmout, this means infinity was specified as the timeout value and the task continues to wait for resource acquisition without timing out.

[Difference with T-Kernel]

E_DISWAI does not exist in error codes. This is because in µT-Kernel, there is no wait-disabled function.

tk_ref_sem

Reference Semaphore Status

[C Language Interface]

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

[Parameters]

ID	semid	Semaphore ID
T_RSEM*	pk_rsem	Address of packet for returning status information

pk_rsem detail:

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

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

[Return Parameters]

ER	ercd	Error code

[Error Codes]

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

[Description]

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

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

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

4.3.2 Event Flag

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

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

tk_cre_flg

Create Event Flag

[C Language Interface]

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

[Parameters]

T_CFLG*	pk_cflg	Information about the event flag to be created

pk_cflg detail:

VP	exinf	Extended information
ATR	flgatr	Event flag attributes
UINT	iflgptn	Initial event flag pattern
UB	dsname[8]	DS object name

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

[Return Parameters]

ID	flgid	Event flag ID
	or	Error Code

[Error Codes]

E_NOMEM	Insufficient memory (memory for control block cannot be allocated)
E_LIMIT	Number of event flags exceeds the system limit
E_RSATR	Reserved attribute (`flgatr` is invalid or cannot be used)
E_PAR	Parameter error (`pk_cflg` is invalid)

[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 bits of the processor's bit width as a group. All operations are performed in units of the processor's bit width.

`exinf` can be used freely by the user to store miscellaneous information about the created event flag.

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

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

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

TA_TFIFO	Tasks are queued in FIFO order
TA_TPRI	Tasks are queued in priority order
TA_WSGL	Waiting for multiple tasks is not allowed (Wait Single Task)
TA_WMUL	Waiting for multiple tasks is allowed (Wait Multiple Task)
TA_DSNAME	Specifies DS object name

When TA_WSGL is specified, multiple tasks cannot be in WAIT state at the same time. Specifying TA_WMUL allows waiting by multiple tasks at the same time.

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

When TA_WSGL is specified, however, since tasks cannot be queued, TA_TFIFO and TA_TPRI have no effect.

When multiple tasks are waiting for an event flag, tasks are checked in order from the head of the queue, and the wait is released for tasks meeting the conditions. The first task to have its WAIT state released is therefore not necessarily the first in the queue. If multiple tasks meet the conditions, the 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 by the debugger to identify objects and is handled only by debugger support functions API, td_ref_dsname and td_set_dsname. For more details, refer to 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  /* allow multiple task waiting     */
#define  TA_DSNAME    0x00000040  /* DS object name is specified     */
```

[Difference with T-Kernel]

TA_NODISWAI does not exist in the attribute for flgatr. This is because, in µT-Kernel, there is no wait-disabled function.

[Porting Guideline]

Note that iflgptn, the member of T_CFLG, is of UINT type, and the bit width can vary depending on the system.

tk_del_flg

Delete Event Flag

[C Language Interface]

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

[Parameters]

```
ID          flgid       Event flag ID
```

[Return Parameters]

```
ER          ercd        Error code
```

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (flgid is invalid or cannot be used)
E_NOEXS	Object does not exist (the event flag specified in flgid does not exist)

[Description]

Deletes the event flag specified in flgid.

Issuing this system call releases the corresponding event flag ID and control block memory space.

This system call is completed normally even if there are tasks waiting for the event flag, but error code E_DLT is returned to each task waiting on this event flag.

tk_set_flg
tk_clr_flg

Set Event Flag
Clear Event Flag

[C Language Interface]

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

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

[Parameters]

For tk_set_flg

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

For tk_clr_flg

ID	flgid	Event flag ID
UINT	clrptn	Bit pattern to be cleared

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (flgid is invalid or cannot be used)
E_NOEXS	Object does not exist (the event flag specified in flgid does not exist)

[Description]

tk_set_flg sets the bits indicated in setptn in the event flag specified in flgid, i.e., a logical sum is taken of the values of the event flag specified in flgid and the values indicated in setptn.

tk_clr_flg clears the bits of the event flag based on the corresponding zero bits of clrptn, i.e., a logical product is taken of the values of the event flag specified in flgid and the values indicated in clrptn.

After event flag values are changed by tk_set_flg, if the condition for releasing the wait state of a task that called tk_wai_flg is met, the WAIT state of that task is cleared, putting it in RUN state or READY state (or SUSPEND state if the waiting task was in WAIT-SUSPEND state).

Issuing tk_clr_flg never results in wait conditions being released for a task waiting for the specified event flag; thus, dispatching never occurs as a result of calling tk_clr_flg.

Nothing will happen to the event flag if all bits of setptn are cleared to 0 with tk_set_flg or if all bits of clrptn are set to 1 with tk_clr_flg. No error will result in either case.

Multiple tasks can wait for a single event flag if that event flag has the TA_WMUL attribute. The event flag in that case has a queue for the waiting tasks. A single tk_set_flg call for such an event flag may result in the release of multiple waiting tasks.

[Porting Guideline]

Note that `setptn` and `clrptn` are of UINT type, and the bit width can vary depending on the system.

tk_wai_flg

Wait Event Flag

[C Language Interface]

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

[Parameters]

ID	flgid	Event flag ID
UINT	waiptn	Wait bit pattern
UINT	wfmode	Wait release condition
TMO	tmout	timeout

[Return Parameters]

ER	ercd	Error code
UINT*	p_flgptn	Event flag bit pattern

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (flgid is invalid or cannot be used)
E_NOEXS	Object does not exist (the event flag specified in flgid does not exist)
E_PAR	Parameter error (waiptn = 0, wfmode is invalid, or tmout ≤ -2)
E_OBJ	Invalid object state (multiple tasks are waiting for an event flag with TA_WSGL attribute)
E_DLT	The object being waited for was deleted (the specified event flag was deleted while waiting)
E_RLWAI	Wait state released (tk_rel_wai received in wait state)
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion or in dispatch disabled state)

[Description]

Waits for the event flag specified in flgid to be set, fulfilling the wait release condition specified in wfmode.

If the event flag specified in flgid already meets the wait release condition set in wfmode, the waiting task continues executing without going to WAIT state.

wfmode is specified as follows.

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

TWF_ANDW	0x00	AND wait condition
TWF_ORW	0x01	OR wait condition
TWF_CLR	0x10	Clear all

TWF_BITCLR	0x20	Clear condition bit only

If TWF_ORW is specified, the issuing task waits for any of the bits specified in waiptn to be set for the event flag specified in flgid (OR wait). If TWF_ANDW is specified, the issuing task will wait for all of the bits specified in waiptn to be set for the event flag specified in flgid (AND wait).

If TWF_CLR specification is not specified, the event flag values will remain unchanged even after the conditions have been satisfied and the task has been released from WAIT state. If TWF_CLR is specified, all bits of the event flag will be cleared to 0 once wait conditions of the waiting task have been met. If TWF_BITCLR is specified, then when the conditions are met and the task is released from WAIT state, only the bits matching the event flag wait release conditions are cleared to 0 (event flag values &= ~wait release conditions).

The return parameter flgptn returns the value of the event flag after the WAIT state of a task has been released due to this system call. If TWF_CLR or TWF_BITCLR was specified, the value before event flag bits were cleared is returned. The value returned by flgptn fulfills the wait release conditions of this system call. The contents of flgptn are indeterminate if the wait is released due to other reasons such as timeout.

A maximum wait time (timeout) can be set in tmout. If the tmout time elapses before the wait release condition is met, the system call terminates, returning timeout error code E_TMOUT. Only positive values can be set in tmout. The time base for tmout (time unit) is the same as that for system time (= 1 ms).

When TMO_POL = 0 is set in tmout, this means 0 was specified as the timeout value, and E_TMOUT is returned without entering WAIT state even if the condition is not met. When TMO_FEVR =(-1) is set in tmout, this means infinity was specified as the timeout value, and the task continues to wait for the condition to be met without timing out.

In the case of a timeout, the event flag bits are not cleared even if TWF_CLR or TWF_BITCLR was specified. Setting waiptn to 0 results in Parameter error E_PAR.

A task cannot execute tk_wai_flg for an event flag having the TA_WSGL attribute while another task is waiting for it. Error code E_OBJ will be returned for the task issuing the subsequent tk_wai_flg, irrespective of whether that task would have gone to WAIT state, i.e., regardless of whether the wait release conditions would be met.

If an event flag has the TA_WMUL attribute, multiple tasks can wait for it at the same time. Waiting tasks can be queued, and the WAIT states of multiple tasks can be released by issuing tk_set_flg just once. If multiple tasks are queued for an event flag with TA_WMUL attribute, the behavior is as follows.

- Tasks are queued in either FIFO or priority order. (Release of wait state does not always start from the head of the queue, however, depending on factors such as waiptn and wfmode settings.)
- If TWF_CLR or TWF_BITCLR was specified by a task in the queue, the event flag is cleared when that task is released from WAIT state.
- Tasks later in the queue than a task specifying TWF_CLR or TWF_BITCLR will see the event flag after it has already been cleared.

If multiple tasks having the same priority are simultaneously released from waiting as a result of tk_set_flg, the order of tasks in the ready queue (precedence) after release will continue to be the same as their original order in the event flag queue.

[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 processor's bit-width bit patterns in combination with tk_set_flg. However, it is not possible to send a message containing only 0s for all bits. Moreover, if the next message is sent before a previous message has been read by tk_wai_flg, the previous message will be lost, i.e., message queuing is not possible.

Since setting waiptn = 0 will result in an E_PAR error, it is guaranteed that the waiptn of tasks waiting for an event flag will not be 0. The result is that if tk_set_flg sets all bits of an event flag to 1, the task at the head of the queue will always be released from waiting no matter what its wait condition is. The ability to have multiple tasks wait for the same event flag is useful in situations like the following. Suppose, for example, that Task B and Task C are waiting for tk_wai_flg calls (2) and (3) until Task A issues (1) tk_set_flg. If multiple tasks are allowed to wait for the event flag, the result will be the same regardless of the order in which system calls (1)(2)(3) are executed (see Figure 4.1). 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.

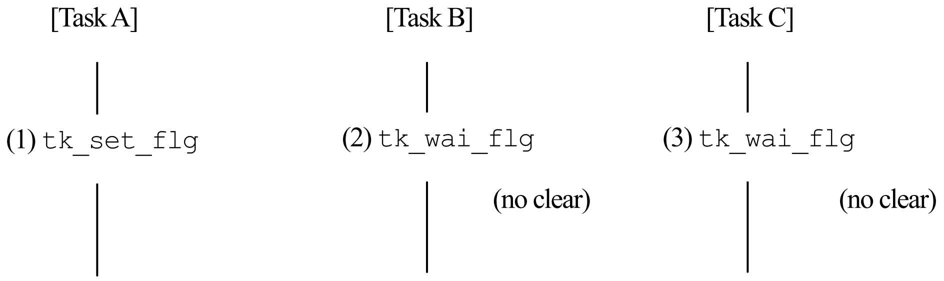


Figure 4.1: Multiple Tasks Waiting for One Event Flag

[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 WAIT state regardless of the subsequent event flag values.

[Difference with T-Kernel]

E_DISWAI does not exist in error codes. This is because in μT-Kernel, there is no wait-disabled function.

[Porting Guideline]

Note that the type of waiptn and the type of the target pointed by p_flgptn are UINT type, and the bit width can vary depending on the system.

tk_ref_flg

Reference Event Flag Status

[C Language Interface]

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

[Parameters]

ID	flgid	Event flag ID
T_RFLG*	pk_rflg	Address of packet for returning status information

[Return Parameters]

ER	ercd	Error code

pk_rflg detail:

VP	exinf	Extended information
ID	wtsk	Waiting task information
UINT	flgptn	Event flag bit pattern

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

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (flgid is invalid or cannot be used)
E_NOEXS	Object does not exist (the event flag specified in flgid does not exist)
E_PAR	Parameter error (the address of the return parameter packet cannot be used)

[Description]

References the status of the event flag specified in flgid, passing in the return parameters the current flag pattern (flgptn), waiting task information (wtsk), and extended information (exinf).

wtsk returns the ID of a task waiting for this event flag. If more than one task is waiting (only when the TA_WMUL was specified), the ID of the first task in the queue is returned. If there are no waiting tasks, wtsk = 0 is returned. If the specified event flag does not exist, error code E_NOEXS is returned.

4.3.3 Mailbox

A mailbox is an object used to achieve synchronization and communication by passing messages in system (shared) memory space. Functions are provided for creating and deleting a mailbox, sending and receiving messages in a mailbox, and referencing the mailbox status. A mailbox is an object identified by an ID number called the mailbox ID.

A mailbox has a message queue for sent messages, and a task queue for tasks waiting to receive messages. At the message sending end (making event notification), messages to be sent go in the message queue. On the message receiving end (waiting for event notification), a task fetches one message from the message queue. If there are no queued messages, the task goes to a state of waiting to receive a message from the mailbox until the next message is sent. Tasks waiting for message receipt from a mailbox are put in the task queue of that mailbox.

Since the contents of messages using this function are in memory shared by the sending and receiving sides, only the start address of a message located in this shared space is actually sent and received. The contents of the messages themselves are not copied. µT-Kernel manages messages in the message queue by means of a link list. At the beginning of a message to be sent, an application program must allocate space for link list use by µT-Kernel. This area is called the message header. The message header and the message body together are called a message packet. When a system call sends a message to a mailbox, the start address of the message packet (`pk_msg`) is passed in a parameter. When a system call receives a message from a mailbox, the start address of the message packet is passed in a return parameter. If messages are assigned a priority in the message queue, the message priority (`msgpri`) of each message must be specified in the message header (see Figure 4.2). The user puts the message contents not at the beginning of the packet but after the header part (`msgcont` in the figure).

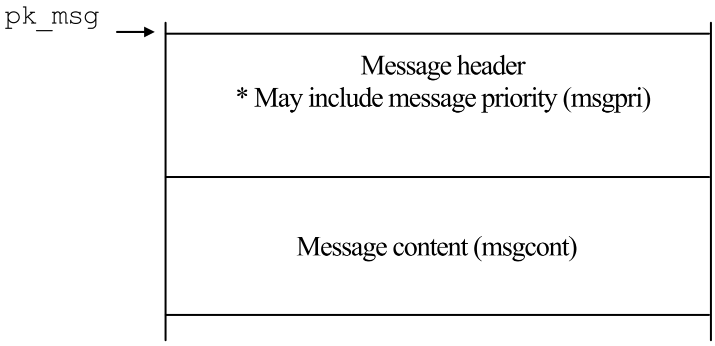

pk_msg
Message header
* May include message priority (msgpri)
Message content (msgcont)

Figure 4.2: Format of Messages Using a Mailbox

µT-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). This restriction applies not only to the direct writing of a message header by an application program, but also to the passing of a header address to µT-Kernel and having µT-Kernel overwrite the message header with the contents.

[Additional Notes]

Since the application program allocates the message header space for this mailbox function, there is no limit on the number of messages that can be queued. A system call sending a message does not enter WAIT state.

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

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

[Difference with T-Kernel]

In µT-Kernel, there is no concept of user space and there is only what T-Kernel calls shared space. Only the start address of the message in shared memory is actually sent and received between the mailboxes. So, it must be noted that the message be placed in shared space instead of user space when used in T-Kernel. In µT-Kernel, awareness of this is not required because everything is placed in shared space.

tk_cre_mbx

Create Mailbox

[C Language Interface]

```
ID mbxid    = tk_cre_mbx ( T_CMBX* pk_cmbx ) ;
```

[Parameters]

T_CMBX*	pk_cmbx	Information about the mailbox to be created

pk_cmbx detail:

VP	exinf	Extended information
ATR	mbxatr	Mailbox attributes
UB	dsname[8]	DS object name

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

[Return Parameters]

ID	mbxid	Mailbox ID
	or	Error Code

[Error Codes]

E_NOMEM	Insufficient memory (memory for the control block or buffer cannot be allocated)
E_LIMIT	Number of mailboxes exceeds the system limit
E_RSATR	Reserved attribute (mbxatr is invalid or cannot be used)
E_PAR	Parameter error (pk_cmbx is invalid)

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

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

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

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

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.

TA_MFIFO and TA_MPRI are used to specify the order of messages in the message queue (messages waiting to be received). If the attribute is TA_MFIFO, messages are ordered by FIFO; TA_MPRI specifies queuing of messages in order of their priority. Message priority is set in a special field in the message packet. Message priority is specified by positive values, with 1 indicating the highest priority and higher numbers indicating successively lower priority. The largest value that can be expressed in the PRI type is the lowest priority. Messages having the same priority are ordered as FIFO.

When TA_DSNAME is specified, dsname is valid and specifies the DS object name. DS object name is used by the debugger to identify objects and is handled only by debugger support functions API, td_ref_dsname and td_set_dsname. For more details, refer to 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_MFIFO    0x00000000  /* manage message queue by FIFO       */
#define  TA_MPRI     0x00000002  /* manage message queue by priority   */
#define  TA_DSNAME   0x00000040  /* DS object name                     */
```

[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.

[Difference with T-Kernel]

TA_NODISWAI does not exist in the attribute for mbxatr. This is because, in µT-Kernel, there is no wait-disabled function.

tk_del_mbx

Delete Mailbox

[C Language Interface]

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

[Parameters]

ID	mbxid	Mailbox ID

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mbxid is invalid or cannot be used)
E_NOEXS	Object does not exist (the mailbox specified in mbxid does not exist)

[Description]

Deletes the mailbox specified in mbxid.

Issuing this system call releases the mailbox ID and control block memory space, etc., associated with the mailbox.

This system call completes normally even if there are tasks waiting for messages in the deleted mailbox, but error code E_DLT is returned to each of the tasks waiting on this mailbox. Even if there are messages in the mailbox, it is deleted without returning an error code.

tk_snd_mbx

Send Message to Mailbox

[C Language Interface]

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

[Parameters]

ID	mbxid	Mailbox ID
T_MSG*	pk_msg	Message packet address

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mbxid is invalid or cannot be used)
E_NOEXS	Object does not exist (the mailbox specified in mbxid does not exist)
E_PAR	Parameter error (pk_msg is a value that cannot be used)

[Description]

Sends the message packet having pk_msg as its start address to the mailbox specified in mbxid. The message packet contents are not copied; only the start address (pk_msg) is passed at the time of message receipt.

If tasks are already waiting for messages in the same mailbox, the WAIT state of the task at the head of the queue is released, and the pk_msg passed to tk_snd_mbx is sent to that task, becoming a parameter returned by tk_rcv_mbx. If there are no tasks waiting for messages in the specified mailbox, the sent message goes in the message queue of that mailbox. In neither case does the task issuing tk_snd_mbx enter WAIT state.

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

```
typedef struct t_msg {
      ?    ?    /* Content is implementation-defined (but fixed length) */
} 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 of variable length.

[Additional Notes]

Messages are sent by `tk_snd_mbx` regardless of the status of the receiving tasks. In other words, message sending is asynchronous. What waits in the queue is not the task itself, but the sent message. So while there are queues of waiting messages and receiving tasks, the sending task does not go to WAIT state.

[Difference with T-Kernel]

In µT-Kernel, there is no concept of user space and there is only what T-Kernel calls shared space. Only the start address of the message in shared memory is actually sent and received between the mailboxes. So, it must be noted that the message be placed in shared space instead of user space when used in T-Kernel. In µT-Kernel, awareness of this is not required because everything is placed in shared space.

tk_rcv_mbx

Receive Message from Mailbox

[C Language Interface]

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

[Parameters]

ID	mbxid	Mailbox ID
TMO	tmout	timeout

[Return Parameters]

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

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mbxid is invalid or cannot be used)
E_NOEXS	Object does not exist (the mailbox specified in mbxid does not exist)
E_PAR	Parameter error (tmout ≤ (-2))
E_DLT	The object being waited for was deleted (the mailbox was deleted while waiting)
E_RLWAI	Wait state released (tk_rel_wai received in wait state)
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion or in dispatch disabled state)

[Description]

Receives a message from the mailbox specified in mbxid. If no messages have been sent to the mailbox (the message queue is empty), the task issuing this system call enters WAIT state and is queued for message arrival. If there are messages in the mailbox, the task issuing this system call fetches the first message in the message queue and passes it in the return parameter pk_msg.

A maximum wait time (timeout) can be set in tmout. If the tmout time elapses before the wait release condition is met (no message arrives), the system call terminates, returning timeout error code E_TMOUT. Only positive values can be set in tmout. The time base for tmout (time unit) is the same as that for system time (= 1 ms).

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

[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).

[Difference with T-Kernel]

E_DISWAI does not exist in error codes. This is because in µT-Kernel, there is no wait-disabled function.

tk_ref_mbx

Reference Mailbox Status

[C Language Interface]

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

[Parameters]

ID	mbxid	Mailbox ID
T_RMBX*	pk_rmbx	Address of packet for returning status information

[Return Parameters]

ER	ercd	Error code

pk_rmbx detail:

VP	exinf	Extended information
ID	wtsk	Waiting task information
T_MSG*	pk_msg	Start address of next message packet to be received

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

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mbxid is invalid or cannot be used)
E_NOEXS	Object does not exist (the mailbox specified in mbxid does not exist)
E_PAR	Parameter error (the return parameter packet address cannot be used)

[Description]

References the status of the mailbox specified in mbxid, passing in the return parameters the next message to be received (the first message in the message queue), waiting task information (wtsk), and extended information (exinf).

wtsk indicates the ID of a task waiting for the mailbox. If there are multiple waiting tasks, the ID of the first task in the queue is returned. If there are no waiting tasks, wtsk = 0 is returned. If the specified mailbox does not exist, error code E_NOEXS is returned.

pk_msg indicates the message that will be received the next time tk_rcv_mbx is issued. If there are no messages in the message queue, pk_msg = NULL is returned. At least one of pk_msg = NULL and wtsk = 0 is always true for this system call.

4.4 Extended Synchronization and Communication Functions

Extended synchronization and communication functions use objects independent of tasks to realize more sophisticated synchronization and communication between tasks. The functions specified here include mutex, message buffer, and rendezvous port functions.

[Difference with T-Kernel]

In µT-Kernel, as there is no wait-disabled function, attribute `TA_NODISWAI` and error code `E_DISWAI` do not exist.

4.4.1 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 tools for managing 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 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, μT-Kernel keeps track of the tasks locking it; and for each task, it keeps track of the mutexes it has locked. Before a task uses a resource, it locks a mutex corresponding to that resource. If the mutex is already locked by another task, the task waits for the mutex to become unlocked. Tasks in mutex lock waiting state are put in the mutex queue. When a task finishes with a resource, it unlocks the mutex. A mutex with `TA_INHERIT` (= 0x02) specified as mutex attribute supports priority inheritance protocol, while one with `TA_CEILING` (= 0x03) specified supports priority ceiling protocol. When a priority ceiling mutex is created, a ceiling priority is assigned to it, indicating the base priority of the task having the highest base priority among the tasks able to lock that mutex. If a task having a higher base priority than the ceiling priority of the mutex tries to lock it, error code `E_ILUSE` is returned. If `tk_chg_pri` is issued in an attempt to set the base priority of a task locking a priority ceiling mutex to a value higher than the ceiling priority of that mutex, `E_ILUSE` is returned by the `tk_chg_pri` system call.

When these protocols are used, unbounded priority inversion is prevented by changing the current priority of a task in a mutex operation. The current priority of the task should be modified to always match the maximum value of the following priorities:

- The task base priority.
- When tasks lock priority inheritance mutexes, the current priority of the task having the highest current priority of the tasks waiting for those mutexes.
- When tasks lock priority ceiling mutexes, the ceiling priority of the mutex having the highest ceiling priority among those mutexes.

Note that when the current priority of a task waiting for a priority inheritance mutex changes as the result of a base priority change brought about by mutex operation or `tk_chg_pri`, it may be necessary to change the current priority of the task locking that mutex. This is called dynamic priority inheritance. Further, if this task is waiting for another priority inheritance mutex, dynamic priority inheritance processing may be necessary also for the task locking that mutex.

When the current priority of a task is changed in connection with a mutex operation, the following processing is performed. If a task with modified priority is in executable state, the task precedence shall be changed according to the modified priority. Among tasks which have the same priority as the modified priority, the task precedence shall be changed to the lowest precedence. Even when the task with modified priority is linked to any priority queue, the task priority in the queue shall be changed according to the modified priority. Among tasks which have the same priority as the modified priority, the task priority shall be changed to the lowest priority. If any mutex locked by the task still remains when the task is completed, all those mutexes shall be unlocked. If multiple mutexes are locked, they are unlocked in the reverse order of allocation. See the description of `tk_unl_mtx` for the specific processing involved.

[Additional Notes]

A `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.

tk_cre_mtx

Create Mutex

[C Language Interface]

```
ID mtxid    = tk_cre_mtx ( T_CMTX *pk_cmtx ) ;
```

[Parameters]

T_CMTX*	pk_cmtx	Information about the mutex to be created

pk_cmtx detail:

VP	exinf	Extended information
ATR	mtxatr	Mutex attributes
PRI	ceilpri	Mutex ceiling priority
UB	dsname[8]	DS object name

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

[Return Parameters]

ID	mtxid	Mutex ID
	or	Error Code

[Error Codes]

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

[Description]

Creates a mutex, assigning to it a mutex ID.

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

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

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

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

When the TA_TFIFO attribute is specified, the order of the mutex task queue is FIFO. If TA_TPRI, TA_INHERIT, or TA_CEILING is specified, tasks are ordered by their priority. TA_INHERIT indicates that priority inheritance protocol is used, and TA_CEILING specifies priority ceiling protocol.

Only when TA_CEILING is specified does ceilpri have validity, setting the mutex ceiling priority.

When TA_DSNAME is specified, dsname is valid and specifies the DS object name. DS object name is used by the debugger to identify objects and is handled only by debugger support functions API, td_ref_dsname and td_set_dsname. For more details, refer to 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_INHERIT   0x00000002  /* priority inheritance protocol   */
#define  TA_CEILING   0x00000003  /* priority ceiling protocol       */
#define  TA_DSNAME    0x00000040  /* DS object name                  */
```

[Difference with T-Kernel]

TA_NODISWAI does not exist in the attribute for mtxatr. This is because, in μT-Kernel, there is no wait-disabled function.

tk_del_mtx

Delete Mutex

[C Language Interface]

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

[Parameters]

ID	mtxid	Mutex ID

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mtxid is invalid or cannot be used)
E_NOEXS	Object does not exist (the mutex specified in mtxid does not exist)

[Description]

Deletes the mutex specified in mtxid.

Issuing this system call releases the mutex ID and control block memory space allocated to the mutex.

This system call completes normally even if there are tasks waiting to lock the deleted mutex, but error code E_DLT is returned to each of the tasks waiting on this mutex.

When a mutex is deleted, a task locking the mutex will have fewer locked mutexes. If the mutex was a priority inheritance mutex or priority ceiling mutex, it is possible that the priority of the task locking it will change as a result of its deletion.

tk_loc_mtx

Lock Mutex

[C Language Interface]

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

[Parameters]

ID	mtxid	Mutex ID
TMO	tmout	timeout

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mtxid is invalid or cannot be used)
E_NOEXS	Object does not exist (the mutex specified in mtxid does not exist)
E_PAR	Parameter error (tmout ≤ (-2))
E_DLT	The object being waited for was deleted (the mutex was deleted while waiting for a lock)
E_RLWAI	Wait state released (tk_rel_wai received in wait state)
E_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)

[Description]

Locks the mutex specified in mtxid. If the mutex can be locked immediately, the task issuing this system call continues executing without entering WAIT state, and the mutex goes to locked status. If the mutex cannot be locked, the task issuing this system call enters WAIT state. That is, the task is put in the queue of this mutex.

A maximum wait time (timeout) can be set in tmout. If the tmout time elapses before the wait release condition is met, the system call terminates, returning timeout error code E_TMOUT. Only positive values can be set in tmout. The time base for tmout (time unit) is the same as that for system time (= 1 ms). When TMO_POL = 0 is set in tmout, this means 0 was specified as the timeout value and E_TMOUT is returned without entering WAIT state even if the resource cannot be locked. When TMO_FEVR =(-1) is set in tmout, this means infinity was specified as the timeout value, and the task continues to wait to until the resource can be locked by it.

If the invoking task has already locked the specified mutex, error code E_ILUSE (multiple lock) is returned.

If the specified mutex is a priority ceiling mutex and the base priority[2] 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 (due to timeout or some other reason), the priority of the task locking that mutex can be lowered to the highest of the following three priorities.

a. The highest priority among the current priorities of tasks waiting to lock the mutex.

b. The highest priority among all the other mutexes locked by the task currently locking this mutex.

c. The base priority of the locking task.

Priority ceiling mutex (TA_CEILING attribute)

If the invoking task obtains a lock and its current priority is lower than the mutex ceiling priority, the priority of the invoking task is raised to the mutex ceiling priority.

[Difference with T-Kernel]

E_DISWAI does not exist in error codes. This is because in µT-Kernel, there is no wait-disabled function.

[2] **Base priority:** The task priority before it is automatically raised by the mutex. This is the priority last set by tk_chg_pri (including while the mutex is locked), or if tk_chg_pri has never been issued, the priority set when the task was created.

tk_unl_mtx

Unlock Mutex

[C Language Interface]

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

[Parameters]

```
ID          mtxid       Mutex ID
```

[Return Parameters]

```
ER          ercd        Error code
```

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mtxid is invalid or cannot be used)
E_NOEXS	Object does not exist (the mutex specified in mtxid does not exist)
E_ILUSE	Illegal use (not a mutex locked by the invoking task)

[Description]

Unlocks the mutex specified in mtxid. If there are tasks waiting to lock the mutex, the WAIT state of the task at the head of the queue for that mutex is released and that task locks the mutex.

If a mutex that was not locked by the invoking task is specified, error code E_ILUSE is returned.

[Additional Notes]

If the unlocked mutex is a priority inheritance mutex or priority ceiling mutex, task priority must be lowered as follows. If, as a result of this operation, the invoking task no longer has any locked mutexes, the invoking task's priority is lowered to its base priority. If the invoking task continues to have locked mutexes after this operation, the invoking task priority is lowered to whichever of the following priority levels is highest.

a. The highest priority of all remaining locked mutexes.

b. The base priority of the task.

If a task terminates (goes to DORMANT state or NON-EXISTENT state) without explicitly unlocking mutexes, all its locked mutexes are automatically unlocked.

tk_ref_mtx

Refer Mutex Status

[C Language Interface]

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

[Parameters]

ID	mtxid	Mutex ID
T_RMTX*	pk_rmtx	Address of packet for returning status information

[Return Parameters]

ER	ercd	Error code

pk_rmtx detail:

VP	exinf	Extended information
ID	htsk	ID of task locking the mutex
ID	wtsk	ID of first task waiting to lock the mutex

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

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mtxid is invalid or cannot be used)
E_NOEXS	Object does not exist (the mutex specified in mtxid does not exist)
E_PAR	Parameter error (the address of the return parameter packet cannot be used)

[Description]

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

htsk returns the ID of the task locking the mutex. If no task is locking it, htsk = 0 is returned.

wtsk indicates the ID of a task waiting to lock the mutex. If there are multiple tasks waiting, the ID of the task at the head of the queue is returned. If no tasks are waiting, wtsk = 0 is returned.

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

4.4.2 Message Buffer

A message buffer is an object for achieving synchronization and communication by the passing of variable-size messages. Functions are provided for creating and deleting a message buffer, sending and receiving messages using a message buffer, and referencing message buffer status. A message buffer is an object identified by an ID number called a message buffer ID.

A message buffer keeps a queue of messages waiting to be sent (send queue) and a queue of tasks waiting for message receipt (receive queue). It also has a message buffer space for holding sent messages. The message sender (the side making event notification) copies to the message buffer a message it wants to send. If there is insufficient space in the message buffer area, the message is queued for sending until enough space is available. A task waiting to send a message to the message buffer is put in the send queue. On the message receipt side (waiting for event notification), one message is fetched from the message buffer. If the message buffer has no messages, the task enters WAIT state until the next message is sent. A task waiting for receipt from a message buffer is put in the receive queue of that message buffer.

Synchronous messaging can be realized by setting the message buffer space size to 0. In that case, both the sending task and receiving task wait for a system call to be invoked by each other, and the message is passed when both sides issue system calls.

[Additional Notes]

The message buffer functioning when the size of the message buffer space is set to 0 is explained here using the example in Figure 4.3. In this example, Task A and Task B run asynchronously.

- If Task A calls `tk_snd_mbf` first, it goes to WAIT state until Task B calls `tk_rcv_mbf`. In this case Task A is put in the message buffer send queue (a).
- On the other hand, if Task B calls `tk_rcv_mbf` first, Task B goes to WAIT state until Task A calls `tk_snd_mbf`. Task B is put in the message buffer receive queue (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; then both tasks go from WAIT state to run state.

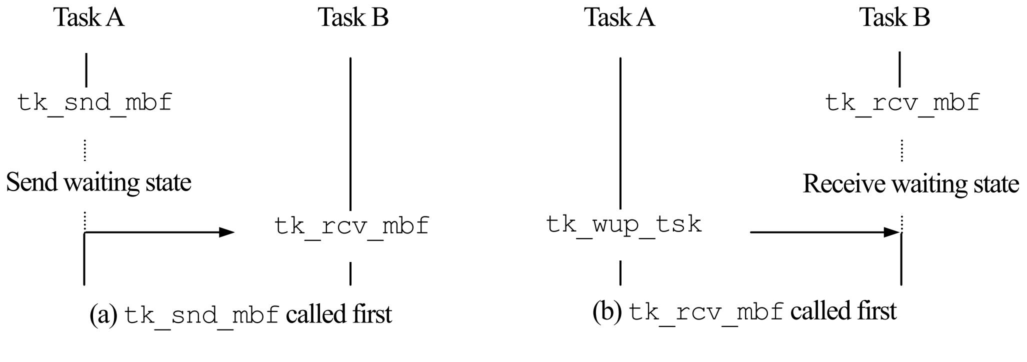


Figure 4.3: Synchronous Communication by Message Buffer

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.

tk_cre_mbf

Create Message Buffer

[C Language Interface]

```
ID mbfid = tk_cre_mbf ( T_CMBF *pk_cmbf ) ;
```

[Parameters]

T_CMBF*	pk_cmbf	Information about the message buffer to be created

pk_cmbf detail:

VP	exinf	Extended information
ATR	mbfatr	Message buffer attributes
W	bufsz	Message buffer size (in bytes)
INT	maxmsz	Maximum message size (in bytes)
UB	dsname[8]	DS object name
VP	bufptr	User buffer pointer

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

[Return Parameters]

ID	mbfid	Message buffer ID
	or	Error Code

[Error Codes]

E_NOMEM	Insufficient memory (memory for control block or ring buffer area cannot be allocated)
E_LIMIT	Number of message buffers exceeds the system limit
E_RSATR	Reserved attribute (mbfatr is invalid or cannot be used)
E_PAR	Parameter error (pk_cmbf is invalid, or bufsz or maxmsz is negative or invalid)

[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. It differs from a mailbox (mbx) in that the contents of the variable-size messages are copied when the message is sent and received. It also has a function for putting the sending task in WAIT state when the buffer is full.

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

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

mbfatr := (TA_TFIFO || TA_TPRI) | [TA_USERBUF] | [TA_DSNAME]

TA_TFIFO	Waiting tasks are queued in FIFO order
TA_TPRI	Waiting tasks are queued in priority order
TA_USERBUF	Indicates that the task uses an area specified by the user as a buffer
TA_DSNAME	Specifies DS object name

The queuing order of tasks waiting for a message to be sent when the buffer is full can be specified in TA_TFIFO or TA_TPRI. If the attribute is TA_TFIFO, tasks are ordered by FIFO, whereas TA_TPRI specifies queuing of tasks in order of their priority setting. Messages themselves are queued in FIFO order only. Tasks waiting for message receipt from a message buffer are likewise queued in FIFO order only.

If TA_USERBUF is specified, bufptr becomes valid and the memory area of bufsz bytes with bufptr at the head is used as a message buffer area. In this case, the message buffer area is not prepared by OS. If TA_USERBUF is not specified, bufptr is ignored, and a message buffer area is allocated by OS.

When TA_DSNAME is specified, dsname is valid and specifies the DS object name. DS object name is used by the debugger to identify objects and is handled only by debugger support functions API, td_ref_dsname and td_set_dsname. For more details, refer to 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_USERBUF   0x00000020  /* User buffer                     */
#define  TA_DSNAME    0x00000040  /* DS object name                  */
```

[Additional Notes]

When there are multiple tasks waiting to send messages, the order in which the messages are sent when buffer space becomes available is always in their queued order. If, for example, a Task A wanting to send a 30-byte message is queued with a Task B wanting to send a 10-byte message, in the order A-B, even if 20 bytes of message buffer space becomes available, Task B never sends its message before Task A.

The ring buffer in which messages are queued also contains information for managing each message. For this reason, the total size of queued messages will ordinarily not be identical to the ring buffer size specified in bufsz. Normally the total message size will be smaller than bufsz. In this sense, bufsz does not strictly represent the total message capacity.

It is possible to create a message buffer with bufsz = 0. In this case communication using the message buffer is completely synchronous between the sending and receiving tasks. That is, if either tk_snd_mbf or tk_rcv_mbf is executed ahead of the other; the task executing the first system call goes to WAIT state. When the other system call is executed, the message is passed (copied) and both tasks resume running.

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

- In Figure 4.4, Task A and Task B operate asynchronously. If Task A arrives at point (1) first and executes tk_snd_mbf(mbfid), Task A goes to send wait state until Task B arrives at point (2). If tk_ref_tsk is issued for Task A in this state, tskwait=TTW_SMBF is returned. If, on the other hand, Task B gets to point (2) first and calls tk_rcv_mbf(mbfid), Task B goes to receive wait state until Task A gets to point (1). If tk_ref_tsk is issued for Task B in this state, tskwait=TTW_RMBF is returned.

- 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.

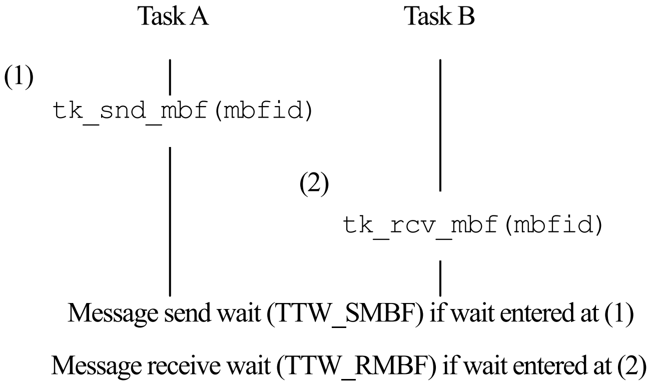

Task A
Task B
(1)
tk_snd_mbf(mbfid)
(2)
tk_rcv_mbf(mbfid)
Message send wait (TTW_SMBF) if wait entered at (1)
Message receive wait (TTW_RMBF) if wait entered at (2)

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

[Difference with T-Kernel]

`TA_NODISWAI` does not exist in the attribute for `mbfatr`. This is because, in µT-Kernel, there is no wait-disabled function.

`TA_USERBUF` and `bufptr` are added.

[Difference with T-Kernel 1.00.00]

`bufsz`, the member of `T_CMBF`, is of W type instead of INT type.

[Porting Guideline]

Note that `maxmsz`, the member of T_CMBF, is of INT type, and the range of values that can be specified may vary depending on the system. For example, in 16 bit environments, the maximum length of message that can be transmitted at a time may be limited to 32767 (approximately 32KB).

There is neither `TA_USERBUF` nor `bufptr` in T-Kernel. So, when using this function, modification is required when porting it to T-Kernel. However, if `bufsz` is set correctly, you can port it by simply deleting `TA_USERBUF` and `bufptr`.

tk_del_mbf

Delete Message Buffer

[C Language Interface]

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

[Parameters]

```
ID          mbfid       Message buffer ID
```

[Return Parameters]

```
ER          ercd        Error code
```

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mbfid is invalid or cannot be used)
E_NOEXS	Object does not exist (the message buffer specified in mbfid does not exist)

[Description]

Deletes the message buffer specified in mbfid.

Issuing this system call releases the corresponding message buffer and control block memory space, as well as the message buffer space.

This system call completes normally even if there are tasks queued in the message buffer for message receipt or message sending, but error code E_DLT is returned to the tasks waiting on this message buffer. Even if there are messages left in the message buffer when it is deleted, the message buffer is deleted anyway. No error code is returned and the messages are discarded.

tk_snd_mbf

Send Message to Message Buffer

[C Language Interface]

```
ER ercd = tk_snd_mbf ( ID mbfid, VP msg, INT msgsz, TMO tmout ) ;
```

[Parameters]

ID	mbfid	Message buffer ID
INT	msgsz	Send message size (in bytes)
VP	msg	Start address of send message packet
TMO	tmout	timeout

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mbfid is invalid or cannot be used)
E_NOEXS	Object does not exist (the message buffer specified in mbfid does not exist)
E_PAR	Parameter error (msgsz ≤ 0, msgsz > maxmsz, value in msg cannot be used, or tmout ≤ (-2))
E_DLT	The object being waited for was deleted (message buffer was deleted while waiting)
E_RLWAI	Wait state released (tk_rel_wai received in wait state)
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion or in dispatch disabled state)

[Description]

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

If msgsz is larger than the maxmsz specified with tk_cre_mbf, error code E_PAR is returned.

If there is not enough available buffer space to accommodate message msg in the message queue, the task issuing this system call goes to send wait state and is queued waiting for buffer space to become available (send queue). Waiting tasks are queued in either FIFO or priority order, depending on the settings specified at message buffer creation with tk_cre_mbf.

A maximum wait time (timeout) can be set in tmout. If the tmout time elapses before the wait release condition is met (before there is sufficient buffer space), the system call terminates, returning timeout error code E_TMOUT.

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

When TMO_POL = 0 is set in tmout, this means 0 was specified as the timeout value, and E_TMOUT is returned without entering WAIT state if there is not enough buffer space.

When TMO_FEVR =(-1) is set in tmout, this means infinity was specified as the timeout value, and the task continues to wait for buffer space to become available, without timing out.

A message of size 0 cannot be sent. When msgsz ≤ 0 is specified, error code E_PAR is returned.

When this system call is invoked from a task-independent portion or in dispatch disabled state, error code E CTX is returned; but in the case of tmout = TMO POL, there may be implementations that enable execution from a task-independent portion or in dispatch disabled state.

[Difference with T-Kernel]

E_DISWAI does not exist in error codes. This is because in µT-Kernel, there is no wait-disabled function.

[Porting Guideline]

Note that msgsz is of INT type and the range of values that can be specified may vary depending on the system. For example, in 16-bit environments, the maximum size of message that can be transmitted at a time may be limited to 32767 (approximately 32KB).

tk_rcv_mbf

Receive Message from Message Buffer

[C Language Interface]

```
INT msgsz = tk_rcv_mbf ( ID mbfid, VP msg, TMO tmout ) ;
```

[Parameters]

ID	mbfid	Message buffer ID
VP	msg	Start address of receive message packet
TMO	tmout	timeout

[Return Parameters]

INT	msgsz	Received message size
	or	Error Code

[Error Codes]

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

[Description]

Receives a message from the message buffer specified in mbfid, putting it in the location specified in msg.

This system call copies the contents of the first queued message in the message buffer specified in mbfid, and copies it to an area of msgsz bytes starting at address msg.

If no message has been sent to the message buffer specified in mbfid (the message queue is empty), the task issuing this system call goes to WAIT state and is put in the receive queue of the message buffer to wait for message arrival. Tasks in the receive queue are ordered by FIFO only.

A maximum wait time (timeout) can be set in tmout. If the tmout time elapses before the wait release condition is met (before a message arrives), the system call terminates, returning timeout error code E_TMOUT.

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

When TMO_POL = 0 is set in tmout, this means 0 was specified as the timeout value, and E_TMOUT is returned without entering WAIT state even if there is no message.

When TMO_FEVR =(-1) is set in tmout, this means infinity was specified as the timeout value, and the task continues to wait for message arrival without timing out.

[Difference with T-Kernel]

E_DISWAI does not exist in error codes. This is because in μT-Kernel, there is no wait disabled function.

tk_ref_mbf

Get Message Buffer Status

[C Language Interface]

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

[Parameters]

ID	mbfid	Message buffer ID
T_RMBF*	pk_rmbf	Address of packet for returning status information

[Return Parameters]

ER	ercd	Error code

pk_rmbf detail:

VP	exinf	Extended information
ID	wtsk	Waiting task information
ID	stsk	Send task information
INT	msgsz	Size of the next message to be received (in bytes)
W	frbufsz	Free buffer size (in bytes)
INT	maxmsz	Maximum message size (in bytes)

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

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mbfid is invalid or cannot be used)
E_NOEXS	Object does not exist (the message buffer specified in mbfid does not exist)
E_PAR	Parameter error (the address of the return parameter packet cannot be used)

[Description]

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

wtsk indicates the ID of the first task waiting to receive a message from the message buffer. The ID of the first task waiting to send to the message buffer is indicated in stsk. If multiple tasks are waiting in the message buffer queues, the ID of the task at the head of the queue is returned. If no tasks are waiting, 0 is returned.

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

The size of the message at the head of the queue (the next message to be received) is returned in msgsz.

If there are no queued messages, msgsz = 0 is returned. A message of size 0 cannot be sent. At least one of msgsz = 0 and wtsk = 0 is always true for this system call. frbufsz indicates the free space in the ring buffer of the message queue. This value indicates the approximate size of messages than can be sent. maxmsz returns the maximum message size as specified with `tk_cre_mbf`.

[Difference with T-Kernel 1.00.00]

`frbufsz`, the member of T_RMBF, is of W type instead of INT type.

4.4.3 Rendezvous Port

Rendezvous is a function for synchronization and communication between tasks, supporting the procedures for making processing requests by one task to another and for returning the processing result to the requesting task. The object for which both of these tasks wait is called a rendezvous port. The rendezvous function is typically used to realize task communication in a client/server model, but can also support more flexible synchronization and communication models.

Functions are provided for creating and deleting a rendezvous port, issuing a processing request to a rendezvous port (call rendezvous), accepting a processing request from a rendezvous port (accept rendezvous), returning the processing result (reply rendezvous), forwarding an accepted processing request to another rendezvous port (forward rendezvous to other port), and referencing rendezvous port status and rendezvous status. A rendezvous port is identified by an ID number called a rendezvous port ID. The task issuing a processing request to a rendezvous port (the client-side task) calls a rendezvous, specifying a message (called a call message) with information about the rendezvous port, the rendezvous conditions, and the processing being requested. The task accepting a processing request on a rendezvous port (the server-side task) accepts the rendezvous, specifying the rendezvous port and rendezvous conditions.

The rendezvous conditions are indicated in a bit pattern. If the bitwise logical AND of the bit patterns on both sides (the rendezvous conditions bit pattern of the task calling a rendezvous for a rendezvous port and the rendezvous conditions bit pattern of the accepting task) is not 0, the rendezvous is established. The state of the task calling the rendezvous is WAIT on rendezvous call until the rendezvous is established. The state of the task accepting a rendezvous is WAIT on rendezvous acceptance until the rendezvous is established.

When a rendezvous is established, a call message is passed from the task that called the rendezvous to the accepting task. The state of the task calling the rendezvous goes to WAIT for rendezvous completion until the requested processing is completed. The task accepting the rendezvous is released from WAIT state and it performs the requested processing. Upon completion of the requested processing, the task accepting the rendezvous passes the result of the processing in a reply message to the calling task and ends the rendezvous. At this point, the WAIT state of the task that called the rendezvous is released. A rendezvous port has separate queues for tasks waiting on rendezvous call (call queue) and tasks waiting on rendezvous acceptance (accept queue). Note, however, that after a rendezvous is established, both tasks that formed the rendezvous are detached from the rendezvous port. In other words, a rendezvous port does not have a queue for tasks waiting for rendezvous completion nor does it keep information about the task performing the requested processing.

µT-Kernel assigns an object number to identify each rendezvous when more than one is established at the same time. The rendezvous object number is called the rendezvous number. How to provide rendezvous number is implementation-defined, but it should at least include information for specifying the task that has called the rendezvous. Numbers must also be uniquely assigned to the extent possible; for example, even if the same task makes multiple rendezvous calls, the first rendezvous and second rendezvous must have different rendezvous numbers assigned.

[Additional Notes]

Rendezvous operation is explained here using the example in Figure 4.5. In this figure, Task A and Task B are running asynchronously.

- If Task A first calls `tk_cal_por`, Task A goes to WAIT state until Task B calls `tk_acp_por`. The state of Task A at this time is WAIT on rendezvous call (a).
- If, on the other hand, Task B first calls `tk_acp_por`, Task B goes to WAIT state until Task A calls `tk_cal_por`. The state of Task B at this time is WAIT on rendezvous acceptance (b).
- A rendezvous is established when both Task A has called `tk_cal_por` and Task B has called `tk_acp_por`. At this time Task A remains in WAIT state while the WAIT state of Task B is released. The state of Task A is WAIT for rendezvous completion.
- The Task A WAIT state is released when Task B calls `tk_rpl_rdv`. Thereafter, both tasks enter a run

state.

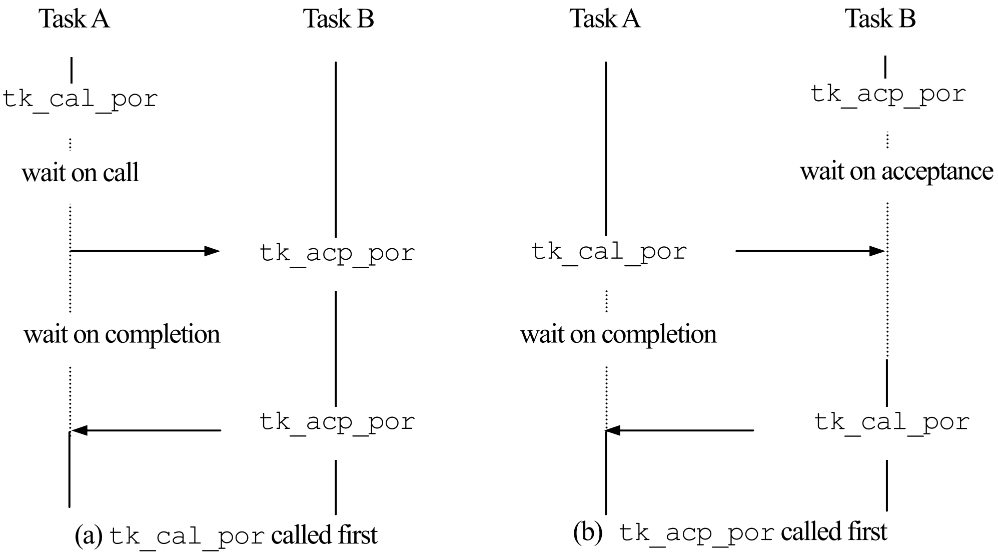


Figure 4.5: Rendezvous Operation

As an example of a specific method for assigning rendezvous object numbers, the ID number of the task calling the rendezvous can go in the low bits of the rendezvous number, with the high bits used for a sequential number.

[Rationale for the Specification]

While it is true that the rendezvous functionality can be achieved through a combination of other synchronization and communication functions, better efficiency and ease of programming are achieved by having a dedicated function for cases where the communication involves an acknowledgment. One advantage of the rendezvous function is that since both tasks wait until message passing is completed, no memory space needs to be allocated for storing messages.

The reason for assigning unique rendezvous numbers even when the same task does the calling is as follows. It is possible that a task, after establishing a rendezvous and going to WAIT state for its completion, will have its WAIT state released due to timeout or forcible release by another task, then again call a rendezvous and have that rendezvous established. If the same number were assigned to both the first and second rendezvous, attempting to terminate the first rendezvous would end up terminating the second rendezvous. If separate numbers are assigned to the two rendezvous and the task in WAIT state for rendezvous completion is made to remember the number of the rendezvous for which it is waiting, error will be returned when the attempt is made to terminate the first rendezvous.

tk_cre_por

Create Port for Rendezvous

[C Language Interface]

```
ID porid = tk_cre_por ( T_CPOR *pk_cpor ) ;
```

[Parameters]

```
T_CPOR*     pk_cpor     Information about the rendezvous port to be created
Pk_cpor detail:
    VP      exinf       Extended information
    ATR     poratr      Rendezvous port attributes
    INT     maxcmsz     Maximum call message size (in bytes)
    INT     maxrmsz     Maximum reply message size (in bytes)
    UB      dsname[8]   DS object name
```

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

[Return Parameters]

```
ID          porid       Port ID
            or          Error Code
```

[Error Codes]

E_NOMEM	Insufficient memory (memory for control block cannot be allocated)
E_LIMIT	Number of rendezvous ports exceeds the system limit
E_RSATR	Reserved attribute (poratr is invalid or cannot be used)
E_PAR	Parameter error (pk_cpor is invalid; maxcmsz or maxrmsz is negative or invalid)

[Description]

Creates a rendezvous port, assigning to it a rendezvous port ID number.

A rendezvous port is an object used as an OS primitive for implementing a rendezvous capability. This system call allocates a control block to the created rendezvous port.

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

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

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

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

When TA_DSNAME is specified, dsname is valid and specifies the DS object name. DS object name is used by the debugger to identify objects and is handled only by debugger support functions API, td_ref_dsname and td_set_dsname. For more details, refer to 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                  */
```

[Difference with T-Kernel]

TA_NODISWAI does not exist in the attribute for poratr. This is because, in µT-Kernel, there is no wait disabled function.

[Porting Guideline]

Note that maxcmsz and maxrmsz, the member of T_CPOR, are of INT type. The range of values that can be specified may vary depending on the system. For example, in 16-bit environments, the maximum length of message that can be transmitted may be limited to 32767 (approximately 32KB).

tk_del_por

Delete Port for Rendezvous

[C Language Interface]

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

[Parameters]

```
ID          porid        Rendezvous port ID
```

[Return Parameters]

```
ER          ercd         Error code
```

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (porid is invalid or cannot be used)
E_NOEXS	Object does not exist (the rendezvous port specified in porid does not exist)

[Description]

Deletes the rendezvous port specified in porid.

Issuing this system call releases the ID number and control block space allocated to the rendezvous port.

This system call completes normally even if there are tasks waiting on rendezvous acceptance (tk_acp_por) or rendezvous port call (tk_cal_por) at the specified rendezvous port, but error code E_DLT is returned to the tasks in WAIT state.

Deletion of a rendezvous port by tk_del_por does not affect tasks for which rendezvous is already established. In this case, nothing is reported to the task accepting the rendezvous (not in WAIT state), and the state of the task calling the rendezvous (WAIT for rendezvous completion) remains unchanged.

When the task accepting the rendezvous issues tk_rpl_rdv, that system call will execute normally even if the port on which the rendezvous was established has been deleted.

tk_cal_por

Call Port for Rendezvous

[C Language Interface]

```
INT rmsgsz = tk_cal_por ( ID porid, UINT calptn, VP msg, INT cmsgsz, TMO tmout ) ;
```

[Parameters]

ID	porid	Rendezvous port ID
UINT	calptn	Call bit pattern (indicating conditions of the caller)
VP	msg	Message packet address
INT	cmsgsz	Call message size (in bytes)
TMO	tmout	Timeout

[Return Parameters]

INT	rmsgsz	Reply message size (in bytes)
	or	Error Code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (porid is invalid or cannot be used)
E_NOEXS	Object does not exist (the rendezvous port specified in porid does not exist)
E_PAR	Parameter error (cmsgsz ≤ 0, cmsgsz > maxcmsz, calptn = 0, value in msg cannot be used, tmout ≤ (-2))
E_DLT	The object being waited for was deleted (the rendezvous port was deleted while waiting)
E_RLWAI	Wait state released (tk_rel_wai received in wait state)
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion or in dispatch disabled state)

[Description]

Issues a rendezvous call for a rendezvous port.

The specific operation of tk_cal_por is as follows. A rendezvous is established if there is a task waiting to accept a rendezvous at the port specified in porid and rendezvous conditions between that task and the task issuing this call overlap. In this case, the task waiting to accept the rendezvous enters READY state while the state of the task issuing tk_cal_por is WAIT for rendezvous completion. The task waiting for rendezvous completion is released from WAIT state when the other (accepting) task executes tk_rpl_rdv. The tk_cal_por system call completes at this time.

If there is no task waiting to accept a rendezvous at the port specified in porid, or if there is a task but conditions for establishing a rendezvous are not satisfied, the task issuing tk_cal_por is placed at the end of the call queue of that port and enters WAIT state on rendezvous call. The order of tasks in the call queue is either FIFO or based on priority, depending on the attributes specified when calling tk_cre_por. The decision on rendezvous establishment is made by checking conditions in the bit patterns acpptn of the accepting task and calptn of the calling task. A rendezvous is established if the bitwise logical AND of these two bit patterns is not 0. Parameter error E_PAR is returned if calptn is 0, since no rendezvous can be established in that case.

When a rendezvous is established, the calling task can send a message (a call message) to the accepting task. The size of the call message is specified in cmsgsz. In this operation, cmsgsz bytes starting at address msg as specified by the calling task when calling tk_cal_por are copied to address msg as specified by the accepting task when calling tk_acp_por.

Similarly, when the rendezvous completes, the accepting task may send a message (reply message) to the calling task. In this operation, the contents of a reply message specified by the accepting task when calling tk_rpl_rdv are copied to address msg as specified by the calling task when calling tk_cal_por. The size of the reply message rmsgsz is set in a return parameter of tk_cal_por parameter. The original contents of the message area passed in msg by tk_cal_por end up being overwritten by the reply message received when tk_rpl_rdv executes.

Note that it is possible that message contents will be destroyed when a rendezvous is forwarded, since an area no larger than maxrmsz starting from the address msg as specified with tk_cal_por is used as a buffer. It is therefore necessary to reserve a memory space of at least maxrmsz starting from msg, regardless of the expected size of the reply message, whenever there is any possibility that a rendezvous requested by tk_cal_por might be forwarded (See the description of tk_fwd_por for details).

Error code E_PAR is returned when cmsgsz exceeds the size maxcmsz specified with tk_cre_por. This error checking is done before a task enters WAIT state on rendezvous call; and if error is detected, the task executing tk_cal_por does not enter WAIT state.

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

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

When TMO_POL = 0 is set in tmout, this means 0 was specified as the timeout value, and E_TMOUT is returned without entering WAIT state if there is no task waiting on a rendezvous at the rendezvous port, or if the rendezvous conditions are not met. When TMO_FEVR =(-1) is set in tmout, this means infinity was specified as the timeout value, and the task continues to wait for a rendezvous to be established without timing out. In any case, tmout indicates only the time allowed for a rendezvous to be established, and does not apply to the time from rendezvous establishment to rendezvous completion.

[Difference with T-Kernel]

E_DISWAI does not exist in error codes. This is because, in μT-Kernel, there is no wait disabled function.

[Porting Guideline]

Note that rmsgsz is of INT type and the range of values that can be specified may vary depending on the system. For example, in 16-bit environments, the maximum size of the message received at a time may be limited up to 32767 (approximately 32KB).

tk_acp_por

Accept Port for Rendezvous

[C Language Interface]

```
INT cmsgsz = tk_acp_por ( ID porid, UINT acpptn, RNO *p_rdvno, VP msg, TMO tmout ) ;
```

[Parameters]

ID	porid	Rendezvous port ID
UINT	acpptn	Accept bit pattern (indicating conditions for acceptance)
VP	msg	Message packet address
TMO	tmout	Timeout

[Return Parameters]

ER	ercd	Error code
RNO*	p_rdvno	Rendezvous number
INT	cmsgsz	Call message size (in bytes)

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (porid is invalid or cannot be used or porid is a rendezvous port of another node)
E_NOEXS	Object does not exist (the rendezvous port specified in porid does not exist)
E_PAR	Parameter error (acpptn = 0, value in msg cannot be used or tmout ≤ (-2))
E_DLT	The object being waited for was deleted (the rendezvous port was deleted while waiting)
E_RLWAI	Wait state released (tk_rel_wai received in wait state)
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion or in dispatch disabled state)

[Description]

Accepts a rendezvous on a rendezvous port.

The specific operation of tk_acp_por is as follows. A rendezvous is established if there is a task queued for a rendezvous call at the port specified in porid and if rendezvous conditions of that task and the task issuing this call overlap. In this case, the task queued for a rendezvous call is removed from the queue, and its state changes from WAIT on rendezvous call to WAIT for rendezvous completion. The task issuing tk_acp_por continues executing.

If there is no task waiting to call a rendezvous at the port specified in porid, or if there is a task but conditions for establishing a rendezvous are not satisfied, the task issuing tk_acp_por will enter WAIT state on rendezvous acceptance for that port. No error results if there is already another task in WAIT state on rendezvous acceptance at this time; the task issuing tk_acp_por is placed in the accept queue. It is possible to conduct multiple rendezvous operations on the same port at the same time. Accordingly, no error results even if the next rendezvous is carried out while another task is still conducting a rendezvous (before tk_rpl_rdv is called for a previously established rendezvous) at the port specified in porid.

The decision on rendezvous establishment is made by checking conditions in the bit patterns acpptn of the accepting task and calptn of the calling task. A rendezvous is established if the bitwise logical AND of these two bit patterns is not 0. If the first task does not satisfy these conditions, each subsequent task in the call queue is checked in succession. If calptn and acpptn are assigned the same non-zero value, rendezvous is established unconditionally. Parameter error E_PAR is returned if acpptn is 0, since no rendezvous can be established in that case. All processing before a rendezvous is established is fully symmetrical on the calling and accepting ends.

When a rendezvous is established, the calling task can send a message (a call message) to the accepting task. The contents of the message specified by the calling task are copied to an area starting from msg specified by the accepting task when tk_acp_por is called. The call message size cmsgsz is passed in a return parameter of tk_acp_por.

A task accepting rendezvous can establish more than one rendezvous at a time. That is, a task that has accepted one rendezvous using tk_acp_por may execute tk_acp_por again before executing tk_rpl_rdv on the first rendezvous. The port specified for the second tk_acp_por call at this time may be the same port as the first rendezvous or a different one. It is even possible for a task already conducting a rendezvous on a given port to execute tk_acp_por again on the same port and conduct multiple rendezvous on the same port at the same time. Of course, the calling tasks will be different in each case. The return parameter rdvno passed by tk_acp_por is used to distinguish different rendezvous when more than one has been established at a given time. It is used as a return parameter by tk_rpl_rdv when a rendezvous completes. It is also passed as a parameter to tk_fwd_por when forwarding a rendezvous.

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

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

When TMO_POL = 0 is set in tmout, this means 0 was specified as the timeout value, and E_TMOUT is returned without entering WAIT state if there is no task waiting for a rendezvous call at the rendezvous port, or if the rendezvous conditions are not met.

When TMO_FEVR =(-1) is set in tmout, this means infinity was specified as the timeout value, and the task continues to wait for a rendezvous to be established without timing out.

[Additional Notes]

The ability to queue tasks accepting rendezvous is useful when multiple servers perform the same processing concurrently. This capability also takes advantage of the task-independent nature of ports. If a task accepting a rendezvous terminates abnormally for some reason before completing its rendezvous (before issuing `tk_rpl_rdv`), the task calling for the rendezvous by issuing `tk_cal_por` will continue waiting indefinitely for rendezvous completion without being released. To avoid such a situation, tasks accepting rendezvous should execute a `tk_rpl_rdv` or `tk_rel_wai` call when they terminate abnormally, as well as notifying the task calling for the rendezvous that the rendezvous ended in error.

`rdvno` contains information specifying the calling task in the rendezvous, but unique numbers should be assigned to the extent possible. Even if different rendezvous are conducted between the same tasks, a different `rdvno` value should be assigned to the first and second rendezvous to avoid problems like the following.

If a task that called `tk_cal_por` and is waiting for rendezvous completion has its WAIT state released by `tk_rel_wai` or by `tk_ter_tsk` + `tk_sta_tsk` or the like, conceivably it may execute `tk_cal_por` a second time, resulting in establishment of a rendezvous. If the same `rdvno` value is assigned to the first rendezvous and the subsequent one, then if `tk_rpl_rdv` is executed for the first rendezvous it will end up terminating the second one. By assigning `rdvno` numbers uniquely and having the task in WAIT state for rendezvous completion remember the number of the expected `rdvno`, it will be possible to detect the error when `tk_rpl_rdv` is called for the first rendezvous.

One possible method of assigning `rdvno` numbers is to put the ID number of the task calling the rendezvous in the low bits of `rdvno`, using the high bits for a sequential number.

The capability of setting rendezvous conditions in `calptn` and `acpptn` can be applied to implement a rendezvous selective acceptance function like the ADA `select` function. A specific processing approach equivalent to an ADA `select` statement (Figure 4.6) is shown in Figure 4.7.

The ADA `select` function is provided only on the accepting end, but it is also possible to implement a select function on the calling end by specifying multiple bits in `calptn`.

[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 the processing required after a rendezvous is established differs for the tasks on each side. That is, whereas the calling task enters WAIT state after the rendezvous is established, the accepting task enters READY state.

[Difference with T-Kernel]

`E_DISWAI` does not exist in error codes. This is because in µT-Kernel, there is no wait disabled function.

[Porting Guideline]

Note that `acpptn` is of UINT type, and the bit width can vary depending on the system.

Note that `cmsgsz` is of INT type and the range of values that can be specified may vary depending on the system. For example, in 16 bit environments, the maximum size of message that can be sent at a time may be limited to 32767 (approximately 32KB).

tk_fwd_por

Forward Rendezvous to Other Port

[C Language Interface]

```
ER ercd = tk_fwd_por ( ID porid, UINT calptn, RNO rdvno, VP msg, INT cmsgsz ) ;
```

[Parameters]

ID	porid	Rendezvous port ID
UINT	calptn	Call bit pattern (indicating conditions of the caller)
RNO	rdvno	Rendezvous number before forwarding
VP	msg	Message packet address
INT	cmsgsz	Call message size (in bytes)

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (porid is invalid or cannot be used, or porid is a rendezvous port of another node)
E_NOEXS	Object does not exist (the rendezvous port specified in porid does not exist)
E_PAR	Parameter error (cmsgsz ≤ 0, cmsgsz > maxcmsz after forwarding, cmsgsz > maxrmsz before forwarding, calptn = 0, or msg has a value that cannot be used)
E_OBJ	Invalid object state (rdvno is invalid, or maxrmsz (after forwarding) > maxrmsz (before forwarding))
E_CTX	Context error (issued from task-independent portion (implementation-dependent error))

[Description]

Forward an accepted rendezvous to another rendezvous port.

The task issuing this system call (here "Task X") must have accepted the rendezvous specified in porid; i.e., this system call can be issued only after executing tk_acp_por. In the discussion that follows, the rendezvous calling task is "Task Y", and the rendezvous number passed in a return parameter by tk_acp_por is rdvno. After tk_fwd_por is issued in this situation, the rendezvous between Task X and Task Y is released, and all processing thereafter is the same as if Task Y had called for a rendezvous on another port (rendezvous port B) passed to this system call in porid.

The specific operations of tk_fwd_por are as follows.

1. The rendezvous specified in rdvno is released.
2. Task Y goes to WAIT state on rendezvous call for the rendezvous port specified in porid. The bit conditions representing the call select conditions in this case are not those given in the calptn specified by Task Y when it called tk_cal_por, but those in the calptn specified by Task X

when it called tk_fwd_por. The state of Task Y goes from WAIT for rendezvous completion back to WAIT on rendezvous call.

3. Then, if a rendezvous for the rendezvous port specified in porid is accepted, a rendezvous is established between the accepting task and Task Y. Naturally, if there is a task already waiting to accept a rendezvous on the rendezvous port specified in porid and the rendezvous conditions are met, executing tk_fwd_por will immediately cause a rendezvous to be established. Here too, as with calptn, the message sent to the accepting task when the rendezvous is established is that specified in tk_fwd_por by Task X, not that specified in tk_cal_por by Task Y.

4. After the new rendezvous has completed, the reply message returned to the calling task by tk_rpl_rdv is copied to the area specified in the msg parameter passed to tk_cal_por by Task Y, not to the area specified in the msg parameter passed to tk_fwd_por by Task X.

Essentially the following situation:

```
Executing tk_fwd_por (porid=portB, calptn=ptnB, msg=mesB)
after tk_cal_por (porid=portA, calptn=ptnA, msg=mesA)
```

is the same as the following:

```
Executing tk_cal_por (porid=portB, calptn=ptnB, msg=mesB).
```

Note that it is not necessary to log the history of rendezvous forwarding. If tk_ref_tsk is executed for a task that has returned to WAIT on rendezvous call due to tk_fwd_por execution, the value returned in tskwait is TTW_CAL. Here wid is the ID of the rendezvous port to which the rendezvous was forwarded.

tk_fwd_por execution completes immediately; in no case does this system call go to a WAIT state. A task issuing tk_fwd_por loses any relationship to the rendezvous port on which the forwarded rendezvous was established, the forwarding destination (the port specified in porid), and the tasks conducting rendezvous on these ports.

Error code E_PAR is returned if cmsgsz is larger than maxcmsz of the rendezvous port after forwarding. This error is checked before the rendezvous is forwarded. If this error occurs, the rendezvous is not forwarded and the rendezvous specified in rdvno is not released.

The send message specified with tk_fwd_por is copied to another memory space (such as the message area specified with tk_cal_por) when tk_fwd_por is executed. Accordingly, even if the contents of the message area specified in the msg parameter passed to tk_fwd_por are changed before the forwarded rendezvous is established, the forwarded rendezvous will not be affected.

When a rendezvous is forwarded by tk_fwd_por, maxrmsz of the rendezvous port after forwarding (specified in porid) must be no larger than maxrmsz of the rendezvous port on which the rendezvous was established before forwarding. If maxrmsz of the rendezvous port after forwarding is larger than maxrmsz of the rendezvous port before forwarding, this means the destination rendezvous port was not suitable, and error code E_OBJ is returned. The task calling the rendezvous readies a reply message receiving area based on the maxrmsz of the rendezvous port before forwarding. If the maximum size for the reply message increases when the rendezvous is forwarded, this may indicate that an unexpectedly large reply message is being returned to the calling rendezvous port, which would cause problems. For this reason, a rendezvous cannot be forwarded to a rendezvous port having a larger maxrmsz.

Similarly, cmsgsz indicating the size of the message sent by tk_fwd_por must be no larger than maxrmsz of the rendezvous port on which the rendezvous was established before forwarding. This is because it is assumed that the message area specified with tk_cal_por will be used as a buffer in implementing tk_fwd_por. If cmsgsz is larger than maxrmsz of the rendezvous port before forwarding, error code E_PAR is returned (See Additional Note for details).

Even when tk_fwd_por and tk_rpl_rdv are issued from the task in dispatch-disabled or interrupt-disabled state, these system calls must behave normally. This capability can be used to perform processing that is inseparable from tk_fwd_por or tk_rpl_rdv.

When, as a result of `tk_fwd_por`, Task Y that was in WAIT state for rendezvous completion reverts to WAIT on rendezvous call, the timeout until rendezvous establishment is always treated as Wait forever (`TMO_FEVR`).

The rendezvous port being forwarded to may be the same port used for the previous rendezvous (the rendezvous port on which the rendezvous specified in `rdvno` was established). In this case, `tk_fwd_por` cancels the previously accepted rendezvous. Even in this case, however, the call message and `calptn` parameters are changed to those passed to `tk_fwd_por` by the accepting task, not those passed to `tk_cal_por` by the calling task.

It is possible to forward a rendezvous that has already been forwarded.

[Additional Notes]

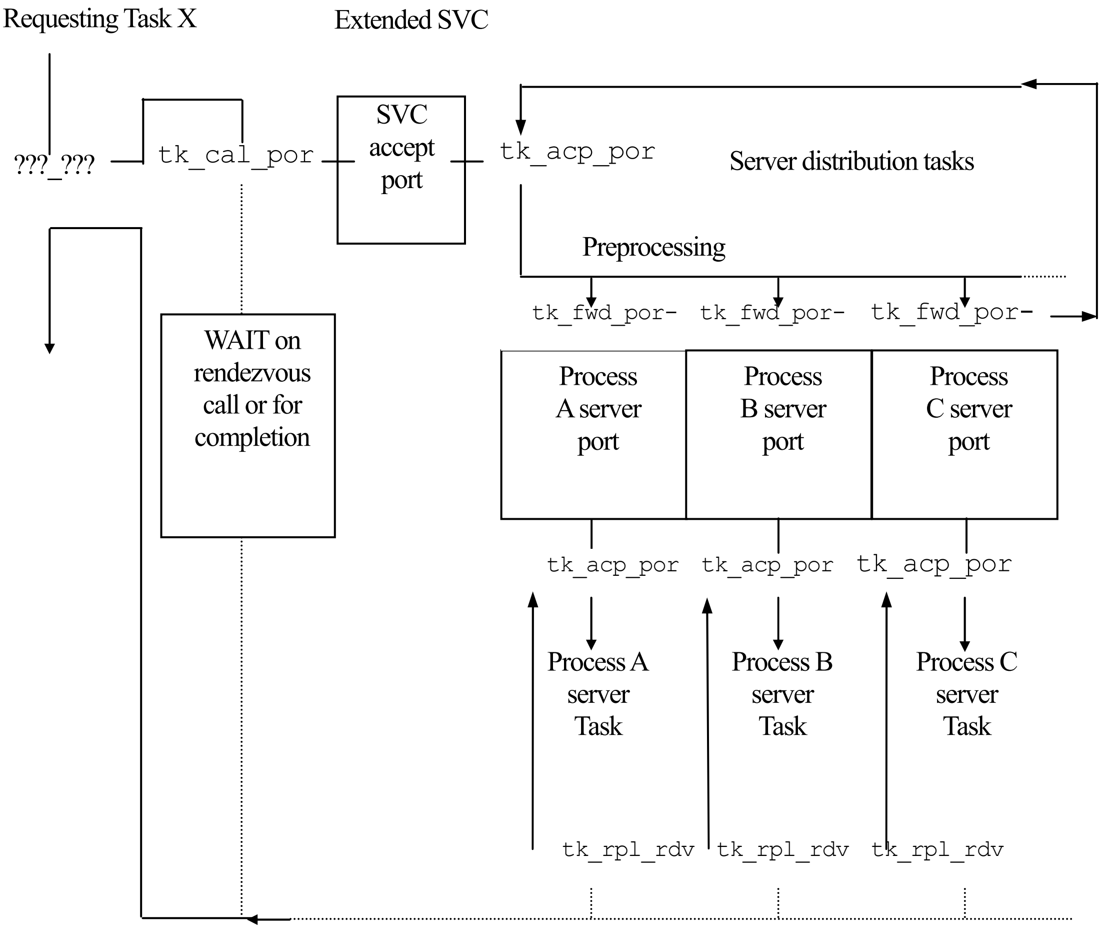
A server task operation using `tk_fwd_por` is illustrated in Figure 4.6.

Generally `tk_fwd_por` is executed by server distribution tasks (tasks for distributing server-accepted processing to other tasks) as shown in Figure 4.6. Accordingly, a server distribution task that has executed `tk_fwd_por` must go on to processing for acceptance of the next request regardless of whether the forwarded rendezvous is established or not. The `tk_fwd_por` message area in this case is used for processing the next request, making it necessary to ensure that changes to the contents of this message area will not affect the previously forwarded rendezvous. For this reason, after `tk_fwd_por` is executed, it must be possible to modify the contents of the message area indicated in `msg` passed to `tk_fwd_por` even before the forwarded rendezvous is established.

In order to fulfill this requirement, in implementation it is allowed to use the message area specified with `tk_cal_por` as a buffer. That is, in the `tk_fwd_por` processing, it is permissible to copy the call messages specified with `tk_fwd_por` to the message area indicated in `msg` when `tk_cal_por` was called, and for the task calling `tk_fwd_por` to change the contents of the message area. When a rendezvous is established, the message placed in the `tk_cal_por` message area is passed to the accepting task, regardless of whether the rendezvous is one that was forwarded from another port.

The following specifications are made to allow this sort of implementation to be used.

- If there is a possibility that a rendezvous requested by `tk_cal_por` may be forwarded, a memory space of at least `maxrmsz` bytes must be allocated starting from `msg` (passed to `tk_cal_por`), regardless of the size of the expected reply message.
- The send message size `cmsgsz` passed to `tk_fwd_por` must be no larger than `maxrmsz` of the rendezvous port before forwarding.
- If a rendezvous is forwarded using `tk_fwd_por`, `maxrmsz` of the destination port rendezvous must not be larger than `maxrmsz` of the port before forwarding.



- 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 4.6: Server Task Operation Using tk_fwd_por

[Rationale for the Specification]

The tk_fwd_por specification is designed to not 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.

[Difference with T-Kernel]

E_DISWAI does not exist in error codes. This is because in µT-Kernel, there is no wait disabled function.

[Porting Guideline]

Note that calptn is of UINT type, and the bit width can vary depending on the processor.

Note that cmsgsz is of INT type and the range of values that can be specified may vary depending on the system. For example, in 16-bit environments, the maximum size of message that can be transmitted at a time may be limited to 32767 (approximately 32KB).

tk_rpl_rdv

Reply Rendezvous

[C Language Interface]

```
ER ercd = tk_rpl_rdv ( RNO rdvno, VP msg, INT rmsgsz ) ;
```

[Parameters]

RNO	rdvno	Rendezvous number
VP	msg	Reply message packet address
INT	rmsgsz	Reply message size (in bytes)

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_PAR	Parameter error (rmsgsz < 0, rmsgsz > maxrmsz, or value in msg cannot be used)
E_OBJ	Invalid object state (rdvno is invalid)
E_CTX	Context error (issued from task-independent portion (implementation-dependent error))

[Description]

Returns a reply to the calling task in the rendezvous, ending the rendezvous.

The task issuing this system call (here "Task X") must be engaged in a rendezvous, i.e., this system call can be issued only after executing tk_acp_por. In the discussion that follows, the rendezvous calling task is "Task Y", and the rendezvous number passed in a return parameter by tk_acp_por is rdvno. When tk_rpl_rdv is executed in this situation, the rendezvous state between Task X and Task Y is released, and the state of Task Y goes from WAIT for rendezvous completion back to READY state. When a rendezvous is ended by tk_rpl_rdv, accepting Task X can send a reply message to calling Task Y. The contents of the message specified by the accepting task are copied to the memory space specified in msg passed by Task Y to tk_cal_por. The size of the reply message rmsgsz is passed as a return parameter of tk_cal_por.

Error code E_PAR is returned if rmsgsz is larger than maxrmsz specified with tk_cre_por. When this error is detected, the rendezvous is not ended and the task that called tk_cal_por remains in WAIT state for rendezvous completion.

Even when tk_fwd_por and tk_rpl_rdv are issued from the task in dispatch-disabled or interrupt-disabled state, these system calls must behave normally. This capability can be used to perform processing that is inseparable from tk_fwd_por or tk_rpl_rdv.

[Additional Notes]

If a task calling a rendezvous aborts for some reason before completion of the rendezvous (before tk_rpl_rdv is executed), the accepting task has no direct way of knowing of the abort. In such a case, error code E_OBJ is returned to the rendezvous accepting task when it executes tk_rpl_rdv.

After a rendezvous is established, tasks are in principle detached from the rendezvous port and have no need to reference information about each other. However, since the value of maxrmsz, used when checking the length of the reply message sent using tk_rpl_rdv, is dependent on the rendezvous port, the task in rendezvous must record this information somewhere. One possible implementation would be to put this information in the TCB of the calling task after it goes to WAIT state, or in another area that can be referenced from the TCB, such as a stack area.

[Rationale for the Specification]

The parameter rdvno is passed to tk_rpl_rdv and tk_fwd_por as information for distinguishing one established rendezvous from another, but the rendezvous port ID (porid) used when establishing a rendezvous is not specified. This is based on the design principle that tasks are no longer related to rendezvous ports after a rendezvous has been established.

Error code E_OBJ rather than E_PAR is returned for an invalid rdvno. This is because rdvno itself is an object indicating the task that called the rendezvous.

[Porting Guideline]

Note that rmsgsz is of INT type and the range of values that can be specified may vary depending on the system. For example, in 16-bit environments, the size of the message that can be received at a time may be limited to 32767 (approximately 32KB).

tk_ref_por

Reference Port Status

[C Language Interface]

```
ER ercd = tk_ref_por ( ID porid, T_RPOR *pk_rpor ) ;
```

[Parameters]

ID	porid	Rendezvous port ID
T_RPOR*	pk_rpor	Start address of packet for returning status information

[Return Parameters]

ER	ercd	Error code

pk_rpor detail:

VP	exinf	Extended information
ID	wtsk	Waiting task information
ID	atsk	Accept task information
INT	maxcmsz	Maximum call message size (in bytes)
INT	maxrmsz	Maximum reply message size (in bytes)

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

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (porid is invalid or cannot be used)
E_NOEXS	Object does not exist (the rendezvous port specified in porid does not exist)
E_PAR	Parameter error (the return parameter packet address cannot be used)

[Description]

References the status of the rendezvous port specified in porid, passing in return parameters information about the accepting task (atsk), information about the task waiting on a rendezvous call (wtsk), maximum message sizes (maxcmsz, maxrmsz), and extended information (exinf).

wtsk indicates the ID of a task in WAIT state on rendezvous call at the rendezvous port. If there is no task waiting on rendezvous call, wtsk = 0 is returned. atsk indicates the ID of a task in WAIT state on rendezvous acceptance at the rendezvous port. If there is no task waiting for rendezvous acceptance, atsk = 0 is returned. If there are multiple tasks waiting on rendezvous call or acceptance at this rendezvous port, the ID of the task at the head of the call queue and accept queue is returned.

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

[Additional Notes]

This system call cannot be used to get information about tasks involved in a currently established rendezvous.

4.5 Memory Pool Management Functions

Memory pool management functions provide software-based management of memory pools and memory block allocation.

There are fixed-size memory pools and variable-size memory pools, which are considered separate objects and require separate sets of system calls for their operation. Memory blocks allocated from a fixed-size memory pool are all of one fixed size, whereas memory blocks from a variable-size memory pool can be of various sizes.

The memory managed by the memory pool management functions is all in system space; there is no μT-Kernel function for managing task space memory.

[Difference with T-Kernel]

In μT-Kernel, since there is no wait-disabled function, attribute `TA_NODISWAI` and error code `E_DISWAI` do not exist.

4.5.1 Fixed-size Memory Pool

A fixed-size memory pool is an object used for dynamic management of fixed-size memory blocks. Functions are provided for creating and deleting a fixed-size memory pool, getting and returning memory blocks in a fixed-size memory pool, and referencing the status of a fixed-size memory pool. A fixed-size memory pool is an object identified by an ID number called a fixed-size memory pool ID.

A fixed-size memory pool has a memory space used as the fixed-size memory pool (called a fixed-size memory pool area or simply memory pool area), and a queue for tasks waiting for memory block allocation. A task wanting to allocate a memory block from a fixed-size memory pool that lacks sufficient available memory space goes to WAIT state for fixed-size memory block until memory blocks are returned to the pool. A task in this state is put in the task queue of the fixed-size memory pool.

[Additional Notes]

When memory blocks of various sizes are needed from fixed-size memory pools, it is necessary to provide multiple memory pools of different sizes.

tk_cre_mpf

Create Fixed-size Memory Pool

[C Language Interface]

```
ID mpfid = tk_cre_mpf ( T_CMPF *pk_cmpf ) ;
```

[Parameters]

T_CMPF*	pk_cmpf	Information about the memory pool to be created

pk_cmpf detail:

VP	exinf	Extended information
ATR	mpfatr	Memory pool attributes
W	mpfcnt	Memory pool block count
W	blfsz	Memory block size (in bytes)
UB	dsname[8]	DS object name
VP	bufptr	User buffer pointer

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

[Return Parameters]

ID	mpfid	Fixed-size memory pool ID
	or	Error Code

[Error Codes]

E_NOMEM	Insufficient memory (memory for control block or memory pool area cannot be allocated)
E_LIMIT	Number of fixed-size memory pools exceeds the system limit
E_RSATR	Reserved attribute (mpfatr is invalid or cannot be used)
E_PAR	Parameter error (pk_cmpf is invalid; mpfcnt or blfsz is negative or invalid)

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

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

```
mpfatr:= (TA_TFIFO || TA_TPRI) | [TA_USERBUF] | [TA_DSNAME]
         | (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_USERBUF`	Indicates that the task uses an area specified by the user as a buffer
`TA_DSNAME`	Specifies DS object name

```
#define  TA_TFIFO     0x00000000  /* manage task queue by FIFO      */
#define  TA_TPRI      0x00000001  /* manage task queue by priority  */
#define  TA_USERBUF   0x00000020  /* User buffer                    */
#define  TA_DSNAME    0x00000040  /* DS object name                 */
#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.

For `TA_RNGn`, a protection level shall be specified to restrict the memory access. However, in µT-Kernel, systems without MMU are assumed. So, all specified protection levels behave the same as protection level 0.

Specification of protection level exists in µT-Kernel only to ensure compatibility with T-Kernel. So, when using only in µT-Kernel, there is no problem if protection level 0 is specified. To create an application available both in µT-Kernel and in T-Kernel, you only need to provide the protection level in T-Kernel.

If `TA_USERBUF` is specified, `bufptr` becomes valid, and the memory area of `mpfcnt*blfsz` bytes with `bufptr` at the head is used as a memory pool area. In this case, the memory pool area is not prepared by the OS. If `TA_USERBUF` is not specified, `bufptr` is ignored, and a memory pool area is allocated by the OS.

When `TA_DSNAME` is specified, `dsname` is valid and specifies the DS object name. DS object name is used by the debugger to identify objects and is handled only by debugger support functions API, `td_ref_dsname` and `td_set_dsname`. For more details, refer to `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.

A system without MMU is assumed for µT-Kernel. However, in order to ensure portability to the T-Kernel, `TA_RNGn` attribute shall be accepted.

It is possible, for example, to treat all `TA_RNGn` as equivalent to `TA_RNG0`; but no error must be returned.

[Difference with T-Kernel]

`TA_NODISWAI` does not exist in the attributes for `mpfatr`. This is because, in µT-Kernel, there is no wait-disabled function.

TA_USERBUF and bufptr have been added.

In µT-Kernel, systems without MMU are assumed. So the functions concerning protection level are not supported. Nevertheless, the reason why TA_RNGn attribute has been retained is to maintain compatibility with the T-Kernel.

[Difference with T-Kernel 1.00.00]

mpfcnt and blfsz, the member of T_CMPF, are of W type instead of INT type.

[Porting Guideline]

There is neither TA_USERBUF nor bufptr in T-Kernel. So, when using this function, modification is required when porting it to T-Kernel. However, if mpfcnt and blfsz are set correctly, you can port it by simply deleting TA_USERBUF and bufptr.

tk_del_mpf

Delete Fixed-size Memory Pool

[C Language Interface]

```
ER ercd = tk_del_mpf ( ID mpfid ) ;
```

[Parameters]

ID	mpfid	Fixed-size memory pool ID

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mpfid is invalid or cannot be used)
E_NOEXS	Object does not exist (the fixed-size memory pool specified in mpfid does not exist)

[Description]

Deletes the fixed-size memory pool specified in mpfid.

No check or notification is made as to whether there are tasks using memory allocated from this memory pool. The system call completes normally even if some blocks have not been returned to the pool.

Issuing this system call releases the memory pool ID number, the control block memory space and the memory pool space itself.

This system call completes normally even if there are tasks waiting for memory block allocation from the deleted memory pool, but error code E_DLT is returned to the tasks in WAIT state.

tk_get_mpf

Get Fixed-size Memory Block

[C Language Interface]

```
ER ercd = tk_get_mpf ( ID mpfid, VP *p_blf, TMO tmout ) ;
```

[Parameters]

ID	mpfid	Fixed-size memory pool ID
TMO	tmout	timeout

[Return Parameters]

ER	ercd	Error code
VP*	p_blf	Memory block start address

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mpfid is invalid or cannot be used)
E_NOEXS	Object does not exist (the fixed-size memory pool specified in mpfid does not exist)
E_PAR	Parameter error (tmout ≤ (-2))
E_DLT	The object being waited for was deleted (the memory pool was deleted while waiting)
E_RLWAI	Wait state released (tk_rel_wai received in wait state)
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion or in dispatch disabled state)

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

When TMO_POL = 0 is set in tmout, this means 0 was specified as the timeout value, and E_TMOUT is returned without entering WAIT state even if memory cannot be allocated.

When TMO_FEVR (= -1) is set in tmout, this means infinity was specified as the timeout value, and the task continues to wait for memory allocation without timing out.

The queuing order of tasks waiting for memory block allocation is either FIFO or in the order of task priority depending on the memory pool attribute.

[Difference with T-Kernel]

E_DISWAI does not exist in error codes. This is because in μT-Kernel, there is no wait-disabled function.

tk_rel_mpf

Release Fixed-size Memory Block

[C Language Interface]

```
ER ercd = tk_rel_mpf ( ID mpfid, VP blf ) ;
```

[Parameters]

ID	mpfid	Fixed-size memory pool ID
VP	blf	Memory block start address

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mpfid is invalid or cannot be used)
E_NOEXS	Object does not exist (the fixed-size memory pool specified in mpfid does not exist)
E_PAR	Parameter error (blf is invalid, or block returned to wrong memory pool)

[Description]

Returns the memory block specified in blf to the fixed-size memory pool specified in mpfid.

Executing tk_rel_mpf may enable memory block acquisition by another task waiting to allocate memory from the memory pool specified in mpfid, releasing the WAIT state of that task.

When a memory block is returned to a fixed-size memory pool, it must be the same fixed-size memory pool from which the block was allocated. If an attempt to return a memory block to a different memory pool is detected, error code E_PAR is returned.

tk_ref_mpf

Reference Fixed-size Memory Pool Status

[C Language Interface]

```
ER ercd = tk_ref_mpf ( ID mpfid, T_RMPF *pk_rmpf ) ;
```

[Parameters]

ID	mpfid	Fixed-size memory pool ID
T_RMPF*	pk_rmpf	Address of packet for returning status information

[Return Parameters]

ER	ercd	Error code

pk_rmpf detail:

VP	exinf	Extended information
ID	wtsk	Waiting task information
W	frbcnt	Free block count

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

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mpfid is invalid or cannot be used)
E_NOEXS	Object does not exist (the fixed-size memory pool specified in mpfid does not exist)
E_PAR	Parameter error (the return parameter packet address cannot be used)

[Description]

References the status of the fixed-size memory pool specified in mpfid, passing in return parameters the current free block count frbcnt, waiting task information (wtsk), and extended information (exinf).

wtsk indicates the ID of a task waiting for memory block allocation from this fixed-size memory pool.

If multiple tasks are waiting for the fixed-size memory pool, the ID of the task at the head of the queue is returned. If there are no waiting tasks, wtsk = 0 is returned.

If the fixed-size memory pool specified with tk_ref_mpf does not exist, error code E_NOEXS is returned.

At least one of frbcnt = 0 and wtsk = 0 is always true for this system call.

[Additional Notes]

Whereas frsz returned by tk_ref_mpl gives the total free memory size in bytes, frbcnt returns the number of unused memory blocks.

[Difference with T-Kernel 1.00.00]

frbcnt, the member of T_RMBF, is of W type instead of INT type.

4.5.2 Variable-size Memory Pool

A variable-size memory pool is an object for dynamically managing memory blocks of any size. Functions are provided for creating and deleting a variable-size memory pool, allocating and returning memory blocks in a variable-size memory pool, and referencing the status of a variable-size memory pool. A variable-size memory pool is an object identified by an ID number called a variable-size memory pool ID.

A variable-size memory pool has a memory space used as the variable-size memory pool (called a variable-size memory pool area or simply memory pool area), and a queue for tasks waiting for memory block allocation. A task wanting to allocate a memory block from a variable-size memory pool that lacks sufficient available memory space goes to WAIT state for variable-size memory block until memory blocks are returned to the pool. A task in this state is put in the task queue of the variable-size memory pool.

[Additional Notes]

When tasks are waiting for memory block allocation from a variable-size memory pool, they are served in queued order. If, for example, Task A requesting a 400-byte memory block from a variable-size memory pool is queued along with Task B requesting a 100-byte block, in A-B order, then even if 200 bytes of space are free, Task B is made to wait until Task A has acquired the requested memory block.

tk_cre_mpl

Create Variable-size Memory Pool

[C Language Interface]

```
ID mplid = tk_cre_mpl ( T_CMPL *pk_cmpl ) ;
```

[Parameters]

T_CMPL*	pk_cmpl	Information about the variable-size memory pool to be created

pk_cmpf detail:

VP	exinf	Extended information
ATR	mplatr	Memory pool attributes
W	mplsz	Memory pool size (in bytes)
UB	dsname[8]	DS object name
VP	bufptr	User buffer pointer

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

[Return Parameters]

ID	mplid	Variable-size memory pool ID
	or	Error Code

[Error Codes]

E_NOMEM	Insufficient memory (memory for control block or memory pool area cannot be allocated)
E_LIMIT	Number of variable-size memory pools exceeds the system limit
E_RSATR	Reserved attribute (mplatr is invalid or cannot be used)
E_PAR	Parameter error (pk_cmpl is invalid, or mplsz is negative or invalid)

[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 allocates a control block to the created memory pool.

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

mplatr indicates system attributes in its low bits and implementation-dependent information in the high bits.

The system attributes part of mplatr is as follows.

    mplatr:= (TA_TFIFO || TA_TPRI) | [TA_USERBUF] | [TA_DSNAME]
              | (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_USERBUF	Indicates that an area specified by the user is used as the buffer
TA_DSNAME	Specifies DS object name

```
#define  TA_TFIFO     0x00000000  /* manage task queue by FIFO       */
#define  TA_TPRI      0x00000001  /* manage task queue by priority   */
#define  TA_USERBUF   0x00000020  /* User buffer                     */
#define  TA_DSNAME    0x00000040  /* DS object name                  */
#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 to acquire memory from a memory pool can be specified in TA_TFIFO or TA_TPRI. If the attribute is TA_TFIFO, tasks are ordered by FIFO, whereas TA_TPRI specifies queuing of tasks in order of their priority setting.

When tasks are queued waiting for memory allocation, memory is allocated in the order of queuing. Even if other tasks in the queue are requesting smaller amounts of memory than the task at the head of the queue, they do not acquire memory blocks before the first task. If, for example, Task A requesting a 400-byte memory block from a variable-size memory pool is queued along with Task B requesting a 100-byte block, in A-B order, then even if 200 bytes of space are free, Task B is made to wait until Task A has acquired the requested memory block.

For TA_RNGn, a protection level shall be specified to restrict memory access. However, in µT-Kernel, systems without MMU are assumed. So, all the specified protection levels behave the same as protection level 0.

If TA_USERBUF is specified, bufptr becomes valid, and the memory area of mplsz bytes with bufptr at the head is used as a memory pool area. In this case, the memory pool area is not prepared by the OS. If TA_USERBUF is not specified, bufptr is ignored, and a memory pool area is allocated by the OS.

When TA_DSNAME is specified, dsname is valid and specifies the DS object name. DS object name is used by the debugger to identify objects and is handled only by debugger support functions API, td_ref_dsname and td_set_dsname. For more details, refer to 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 WAIT state forcibly released or if a different task becomes the first in the queue as a result of a change in task priority, allocation of memory to that task is attempted. If memory can be allocated, the WAIT state of that task is released.

In this way, it is possible under some circumstances for memory allocation to take place and task WAIT state to be released even when memory is not released by tk_rel_mpl.

Systems without MMU are assumed for µT-Kernel. However, in order to ensure portability to the T-Kernel, the TA_RNGn attribute shall be accepted.

It is possible, for example, to treat all TA_RNGn as equivalent to TA_RNG0 but no error must be returned.

[Rationale for the Specification]

The capability of creating multiple memory pools can be used for memory allocation as needed for error handling or in emergencies, etc.

[Difference with T-Kernel]

`TA_NODISWAI` does not exist in the attribute for `mplatr`. This is because in μT-Kernel, there is no wait disabled function.

`TA_USERBUF` and `bufptr` have been added.

In μT-Kernel, systems without MMU are assumed. So the functions relating to protection levels are not supported. However, the `TA_RNGn` attribute has been retained to maintain compatibility with the T-Kernel.

[Difference with T-Kernel 1.00.00]

`mplsz`, the member of `T_CMPL`, is of W type instead of INT type.

[Porting Guideline]

There is neither `TA_USERBUF` nor `bufptr` in T-Kernel. So, when using this function, modification is required when porting it to the T-Kernel. However if `mplsz` is set correctly, you can port it by simply deleting `TA_USERBUF` and `bufptr`.

tk_del_mpl

Delete Variable-size Memory Pool

[C Language Interface]

```
ER ercd = tk_del_mpl ( ID mplid ) ;
```

[Parameters]

ID	mplid	Variable-size memory pool ID

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mplid is invalid or cannot be used)
E_NOEXS	Object does not exist (the variable-size memory pool specified in mplid does not exist)

[Description]

Deletes the variable-size memory pool specified in mplid.

No check or notification is made as to whether there are tasks using memory allocated from this memory pool. The system call completes normally even if some blocks have not been returned to the pool.

Issuing this system call releases the memory pool ID number, the control block memory space and the memory pool space itself.

This system call completes normally even if there are tasks waiting for memory block allocation from the deleted memory pool, but error code E_DLT is returned to the tasks in WAIT state.

tk_get_mpl

Get Variable-size Memory Block

[C Language Interface]

```
ER ercd = tk_get_mpl ( ID mplid, W blksz, VP *p_blk, TMO tmout ) ;
```

[Parameters]

ID	mplid	Variable-size memory pool ID
INT	blksz	Memory block size (in bytes)
TMO	tmout	Timeout

[Return Parameters]

ER	ercd	Error code
VP*	p_blk	Block start address

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mplid is invalid or cannot be used)
E_NOEXS	Object does not exist (the variable-size memory pool specified in mplid does not exist)
E_PAR	Parameter error (tmout ≤ (-2))
E_DLT	The object being waited for was deleted (the memory pool was deleted while waiting)
E_RLWAI	Wait state released (tk_rel_wai received in wait state)
E_TMOUT	Polling failed or timeout
E_CTX	Context error (issued from task-independent portion or in dispatch disabled state)

[Description]

Gets a memory block of size blksz (bytes) from the variable-size memory pool specified in mplid. The start address of the allocated memory block is returned in blk.

The allocated memory is not cleared to zero, and the memory block contents are indeterminate.

If memory cannot be allocated, the task issuing this system call enters WAIT state.

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

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

When TMO_POL = 0 is set in tmout, this means 0 was specified as the timeout value, and E_TMOUT is returned without entering WAIT state even if memory cannot be allocated.

When TMO_FEVR (= -1) is set in tmout, this means infinity was specified as the timeout value, and the task continues to wait for memory allocation without timing out.

The queuing order of tasks waiting for memory block allocation is either FIFO or task priority order, depending on the memory pool attribute.

[Difference with T-Kernel]

`E_DISWAI` does not exist in error codes. This is because in µT-Kernel, there is no wait disabled function.

[Difference with T-Kernel 1.00.00]

The type of parameter `blksz` is W instead of INT.

tk_rel_mpl

Release Variable-size Memory Block

[C Language Interface]

```
ER ercd = tk_rel_mpl ( ID mplid, VP blk ) ;
```

[Parameters]

ID	mplid	Variable-size memory pool ID
VP	blk	Memory block start address

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mplid is invalid or cannot be used)
E_NOEXS	Object does not exist (the variable-size memory pool specified in mplid does not exist)
E_PAR	Parameter error (blk is invalid, or block returned to wrong memory pool)

[Description]

Returns the memory block specified in blk to the variable-size memory pool specified in mplid.

Executing tk_rel_mpl may enable memory block acquisition by another task waiting to allocate memory from the memory pool specified in mplid, releasing the WAIT state of that task.

When a memory block is returned to a variable-size memory pool, it must be the same variable-size memory pool from which the block was allocated. If an attempt to return a memory block to a different memory pool is detected, error code E_PAR is returned. Whether or not this error detection is carried out is implementation-dependent.

[Additional Notes]

When memory is returned to a variable-size memory pool in which multiple tasks are queued, multiple tasks may be released at the same time depending on the amount of memory returned and the requested memory sizes. The task precedence among tasks of the same priority after their WAIT state is released in such a case is the order in which they were queued.

tk_ref_mpl

Reference Variable-size Memory Pool Status

[C Language Interface]

```
ER ercd = tk_ref_mpl ( ID mplid, T_RMPL *pk_rmpl ) ;
```

[Parameters]

ID	mplid	Variable-size memory pool ID
T_RMPL*	pk_rmpl	Address of packet for returning status information

[Return Parameters]

ER	ercd	Error code

pk_rmpl details:

VP	exinf	Extended information
ID	wtsk	Waiting task information
W	frsz	Free memory size (in bytes)
W	maxsz	Maximum memory space size (in bytes)

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

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (mplid is invalid or cannot be used)
E_NOEXS	Object does not exist (the variable-size memory pool specified in mplid does not exist)
E_PAR	Parameter error (the address of the return parameter packet cannot be used)

[Description]

References the status of the variable-size memory pool specified in mplid, passing in return parameters the size of the total free space (frsz), the maximum size of memory immediately available (maxsz), waiting task information (wtsk), and extended information (exinf).

wtsk indicates the ID of a task waiting for memory block allocation from this variable-size memory pool. If multiple tasks are waiting for the variable-size memory pool, the ID of the task at the head of the queue is returned. If there are no waiting tasks, wtsk = 0 is returned. If the variable-size memory pool specified with tk_ref_mpl does not exist, error code E_NOEXS is returned.

[Difference with T-Kernel 1.00.00]

frsz and maxsz , the member of T_RMPL, are of W type instead of INT type.

4.6 Time Management Functions

Time management functions are for performing time-dependent processing. They include functions for system time management, cyclic handlers, and alarm handlers.

The general name used here for cyclic handlers and alarm handlers is time event handlers.

4.6.1 System Time Management

System time management functions are for manipulating system time. Functions are provided for reading and setting system clock, and for reading system operating time.

tk_set_tim

Set Time

[C Language Interface]

```
ER ercd = tk_set_tim ( SYSTIM *pk_tim ) ;
```

[Parameters]

```
SYSTIM*     pk_tim      Address of current time packet
pk_tim detail:
    SYSTIM  systim      Current system time
```

[Return Parameters]

```
ER          ercd        Error code
```

[Error Codes]

E_OK	Normal completion
E_PAR	Parameter error (pk_tim is invalid, or time setting is invalid)

[Description]

Sets the system clock to the value specified in systim.

System time is expressed as cumulative milliseconds from 0:00:00 (GMT), January 1, 1985.

[Additional Notes]

The relative time specified in RELTIM or TMO does not change even if the system clock is changed by calling tk_set_tim during system operation. For example, if a timeout is set to elapse in 60 seconds and the system clock is advanced by 60 seconds by tk_set_tim while waiting for the timeout, the timeout occurs not immediately but 60 seconds after it was set. Instead, tk_set_tim changes the system time at which the timeout occurs.

tk_get_tim

Get Time

[C Language Interface]

```
ER ercd = tk_get_tim ( SYSTIM *pk_tim ) ;
```

[Parameters]

```
SYSTIM*    pk_tim     Address of a packet to return current time
```

[Return Parameters]

```
ER         ercd       Error code
```

```
pk_tim detail:
    SYSTIM systim     Current system time
```

[Error Codes]

E_OK	Normal completion
E_PAR	Parameter error (pk_tim is invalid)

[Description]

Reads the current value of the system clock and returns it in systim.

System time is expressed as cumulative milliseconds from 0:00:00 (GMT), January 1, 1985.

tk_get_otm

Get Operating Time

[C Language Interface]

```
ER ercd = tk_get_otm ( SYSTIM *pk_tim ) ;
```

[Parameters]

```
SYSTIM*     pk_tim      Address of packet for returning operating time
```

```
pk_tim detail:
    SYSTIM  opetim      System operating time
```

[Return Parameters]

```
ER          ercd        Error code
```

[Error Codes]

E_OK	Normal completion
E_PAR	Parameter error (pk_tim is invalid)

[Description]

Gets the system operating time (up time).

System operating time, unlike system time, indicates the length of time elapsed linearly since the system was started. It is not affected by clock settings made by tk_set_tim.

System operating time must have the same precision as system time.

4.6.2 Cyclic Handler

A cyclic handler is a time event handler started at regular intervals. Cyclic handler functions are provided for creating and deleting a cyclic handler, activating and deactivating a cyclic handler operation, and referencing cyclic handler status. A cyclic handler is an object identified by an ID number called a cyclic handler ID.

The time interval at which a cyclic handler is started (cycle time) and the cycle phase are specified for each cyclic handler when it is created. When a cyclic handler operation is requested, µT-Kernel determines the time at which the cyclic handler should be started next based on the cycle time and cycle phase set for it. When a cyclic handler is created, the time when it is to be started next is the time of its creation plus the cycle phase. When the time comes to start a cyclic handler, `exinf`, containing extended information about the cyclic handler, is passed to it as a starting parameter. The time when the cyclic handler is started plus its cycle time becomes the next start time. Sometimes when a cyclic handler is activated, the next start time will be newly set.

The cycle phase of the cyclic handler is less than or equal to cycle time in principle, but if the time set for the cycle phase is longer than the cycle time, the cyclic handler is invoked first after the time specified for the cycle phase has passed. For example, if the cycle time is 100 ms and the cycle phase is 200 ms, the cyclic handler is first invoked after 200ms has elapsed, and then invoked after 200+100*(n-1) ms of time has elapsed.

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 is active or inactive upon creation 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 when 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.

tk_cre_cyc

Create Cyclic Handler

[C Language Interface]

```
ID cycid = tk_cre_cyc ( T_CCYC *pk_ccyc ) ;
```

[Parameters]

T_CCYC* pk_ccyc Address of cyclic handler definition packet

pk_ccyc detail:

VP	exinf	Extended information
ATR	cycatr	Cyclic handler attributes
FP	cychdr	Cyclic handler address
RELTIM	cyctim	Cycle time
RELTIM	cycphs	Cycle phase
UB	dsname[8]	DS object name

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

[Return Parameters]

ID	cycid	Cyclic handler ID
	or	Error Code

[Error Codes]

E_NOMEM	Insufficient memory (memory for control block cannot be allocated)
E_LIMIT	Number of cyclic handlers exceeds the system limit
E_RSATR	Reserved attribute (cycatr is invalid or cannot be used)
E_PAR	Parameter error (pk_ccyc, cychdr, cyctim, or cycphs is invalid or cannot be used)

[Description]

Creates a cyclic handler, assigning to it a cyclic handler ID. A cyclic handler is a handler running at specified intervals as a task-independent portion.

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

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

cycatr := (TA_ASM || TA_HLNG) | [TA_STA] | [TA_PHS] | [TA_DSNAME]

TA_ASM The handler is written in assembly language

TA_HLNG	The handler is written in a 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 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 the function. The cyclic handler takes the following format when the TA_HLNG attribute is specified.

```
void cychdr( VP exinf )
{
      /*
      Processing
      */

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

The cyclic handler format when the TA_ASM attribute is specified is implementation-dependent, but exinf must be passed to the handler in a parameter when it starts.

cycphs indicates the length of time until the cyclic hander is initially started after being created by tk_cre_cyc. Thereafter, it is started periodically at the interval set in cyctim. If zero is specified for cycphs, the cyclic handler starts immediately after it is created. Zero cannot be specified for cyctim. The starting of the cyclic handler for the nth time occurs after at least 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 RUN state up to that time to go to another state, with a different task going to RUN 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 occurring 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 WAIT 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 by the debugger to identify objects and is handled only by debugger support functions API, `td_ref_dsname` and `td_set_dsname`. For more details, refer to `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 an implementation-dependent matter whether to have them run serially (after one handler exits, another starts) or nested (one handler operation is suspended, another runs, and when that one finishes, the previous one resumes). In either case, since time event handlers and interrupt handlers run as task-independent portions, the principle of delayed dispatching applies.

tk_del_cyc

Delete Cyclic Handler

[C Language Interface]

```
ER ercd = tk_del_cyc ( ID cycid ) ;
```

[Parameters]

```
ID          cycid        Cyclic handler ID
```

[Return Parameters]

```
ER          ercd         Error code
```

[Error Codes]

`E_OK`	Normal completion
`E_ID`	Invalid ID number (`cycid` is invalid or cannot be used)
`E_NOEXS`	Object does not exist (the cyclic handler specified in `cycid` does not exist)

[Description]

Deletes the cyclic handler specified in `cycid`.

tk_sta_cyc

Start Cyclic Handler

[C Language Interface]

```
ER ercd = tk_sta_cyc ( ID cycid ) ;
```

[Parameters]

ID	cycid	Cyclic handler ID

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (cycid is invalid or cannot be used)
E_NOEXS	Object does not exist (the cyclic handler specified in cycid does not exist)

[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.

tk_stp_cyc

Stop Cyclic Handler

[C Language Interface]

```
ER ercd = tk_stp_cyc ( ID cycid ) ;
```

[Parameters]

```
ID          cycid        Cyclic handler ID
```

[Return Parameters]

```
ER          ercd         Error code
```

[Error Codes]

`E_OK`	Normal completion
`E_ID`	Invalid ID number (`cycid` is invalid or cannot be used)
`E_NOEXS`	Object does not exist (the cyclic handler specified in `cycid` does not exist)

[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).

tk_ref_cyc

Reference Cyclic Handler Status

[C Language Interface]

```
ER ercd = tk_ref_cyc ( ID cycid, T_RCYC *pk_rcyc ) ;
```

[Parameters]

```
ID         cycid      Cyclic handler ID
T_RCYC*    pk_rcyc    Address of packet for returning status information
```

[Return Parameters]

```
ER          ercd      Error code
pk_rcyc detail:
    VP      exinf     Extended information
    RELTIM  lfttim    Time remaining until the next start time
    UINT    cycstat   Cyclic handler activation state
```

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

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (cycid is invalid or cannot be used)
E_NOEXS	Object does not exist (the cyclic handler specified in cycid does not exist)
E_PAR	Parameter error (the address of the return parameter packet cannot be used)

[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`)

TCYC_STP	The cyclic handler is inactive
TCYC_STA	The cyclic handler is active

```
#define   TCYC_STP   0x00   /* cyclic handler is inactive    */
#define   TCYC_STA   0x01   /* cyclic handler is active      */
```

If the cyclic handler specified in `cycid` does not exist, error code `E_NOEXS` is returned.

4.6.3 Alarm Handler

An alarm handler is a time event handler that starts at a specified time. Functions are provided for creating and deleting an alarm handler, activating and deactivating the alarm handler, and referencing the alarm handler status. An alarm handler is an object identified by an ID number called an alarm handler ID.

The time at which an alarm handler starts (called the alarm time) can be set independently for each alarm handler. When the alarm time arrives, `exinf`, containing extended information about the alarm handler, is passed to it as a starting parameter.

After an alarm handler is created, initially it has no alarm time set and is in inactive state. The alarm time is set when the alarm handler is activated by calling `tk_sta_alm`, as relative time from the time that the system call is executed. When `tk_stp_alm` is called deactivating the alarm handler, the alarm time setting is canceled. Similarly, when the alarm time arrives and the alarm handler runs, the alarm time is canceled and the alarm handler becomes inactive.

tk_cre_alm

Create Alarm Handler

[C Language Interface]

```
ID almid = tk_cre_alm ( T_CALM *pk_calm ) ;
```

[Parameters]

T_CALM*	pk_calm	Address of packet for alarm handler definition

pk_calm detail:

VP	exinf	Extended information
ATR	almatr	Alarm handler attributes
FP	almhdr	Alarm handler address
UB	dsname[8]	DS object name

[Return Parameters]

ID	almid	Alarm handler ID
	or	Error Code

[Error Codes]

E_NOMEM	Insufficient memory (memory for control block cannot be allocated)
E_LIMIT	Number of alarm handlers exceeds the system limit
E_RSATR	Reserved attribute (almatr is invalid or cannot be used)
E_PAR	Parameter error (pk_calm, almatr or almhdr is invalid or cannot be used)

[Description]

Creates an alarm handler, assigning to it an alarm handler ID. An alarm handler is a handler running at the specified time as a task-independent portion.

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

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

almatr := (TA_ASM || TA_HLNG) | [TA_DSNAME]

TA_ASM	The handler is written in assembly language
TA_HLNG	The handler is written in a high-level language
TA_DSNAME	Specifies DS object name

```
#define  TA_ASM    0x00000000  /* assembly program               */
#define  TA_HLNG   0x00000001  /* high-level language program    */
#define  TA_DSNAME 0x00000040  /* DS object name                 */
```

almatr specifies the alarm handler start address.

When the TA_HLNG attribute is specified, the alarm handler is started via a high-level language support routine. The high-level language support routine takes care of saving and restoring register values. The alarm handler terminates by a simple return from the function. The alarm handler takes the following format when the TA_HLNG attribute is specified.

```
void almhdr( VP exinf )
{
        /*
        Processing
        */

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

The alarm handler format when the TA_ASM attribute is specified is implementation-dependent, but exinf must be passed to the handler in a parameter when it starts.

Even if a system call is invoked from an alarm handler and this causes the task in RUN state up to that time to go to another state, with a different task going to RUN 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; dispatching takes place only when the alarm handler terminates. In other words, a dispatch request occurring while an alarm handler is running is not processed immediately, but is delayed until the alarm handler terminates. This is called delayed dispatching.

An alarm handler runs as a task-independent portion. As such, it is not possible to call in an alarm handler a system call that can enter WAIT state, or one that is intended for the invoking task.

When TA_DSNAME is specified, dsname is valid and specifies the DS object name. DS object name is used by the debugger to identify objects and is handled only by debugger support functions API, td_ref_dsname and td_set_dsname. For more details, refer to 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 run simultaneously, whether handlers are serially invoked (one handler runs after another handler exits) or these handlers are nested to be invoked (one handler runs after another handler is suspended, and the other resumes after the one exits) or not is implementation-defined. In either case, since time event handlers and interrupt handlers run as task-independent portions, the principle of delayed dispatching applies.

tk_del_alm

Delete Alarm Handler

[C Language Interface]

```
ER ercd = tk_del_alm ( ID almid ) ;
```

[Parameters]

```
ID          almid       Alarm handler ID
```

[Return Parameters]

```
ER          ercd        Error code
```

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (almid is invalid or cannot be used)
E_NOEXS	Object does not exist (the alarm handler specified in almid does not exist)

[Description]

Deletes the alarm handler specified in almid.

`tk_sta_alm`

Start Alarm Handler

[C Language Interface]

```
ER ercd = tk_sta_alm ( ID almid, RELTIM almtim ) ;
```

[Parameters]

```
ID          almid       Alarm handler ID
RELTIM      almtim      Alarm handler start time (alarm time)
```

[Return Parameters]

```
ER          ercd        Error code
```

[Error Codes]

`E_OK`	Normal completion
`E_ID`	Invalid ID number (`almid` is invalid or cannot be used)
`E_NOEXS`	Object does not exist (the alarm handler specified in `almid` does not exist)

[Description]

Sets the alarm time of the alarm handler specified in `almid` to the time given in `almtim`, putting the alarm handler in active state. `almtim` is specified as relative time from the time of calling `tk_sta_alm`.

After the time specified in `almtim` has elapsed, the alarm handler starts. If the alarm handler is already active when this system call is invoked, the existing `almtim` setting is canceled and the alarm handler is activated again with the alarm time specified here.

If `almtim` = 0 is set, the alarm handler starts as soon as it is activated.

tk_stp_alm

Stop Alarm Handler

[C Language Interface]

```
ER ercd = tk_stp_alm ( ID almid ) ;
```

[Parameters]

ID	almid	Alarm handler ID

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (almid is invalid or cannot be used)
E_NOEXS	Object does not exist (the alarm handler specified in almid does not exist)

[Description]

Cancels the alarm time of the alarm handler specified in almid, putting it in inactive state. If it was already in the inactive state, this system call has no effect (no operation).

tk_ref_alm

Reference Alarm Handler Status

[C Language Interface]

```
ER ercd = tk_ref_alm ( ID almid, T_RALM *pk_ralm ) ;
```

[Parameters]

ID	almid	Alarm handler ID
T_RALM*	pk_ralm	Address of packet for returning status information

[Return Parameters]

ER	ercd	Error code

pk_ralm detail:

VP	exinf	Extended information
RELTIM	lfttim	Time remaining until the handler starts
UINT	almstat	Alarm handler activation state

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

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (almid is invalid or cannot be used)
E_NOEXS	Object does not exist (the alarm handler specified in almid does not exist)
E_PAR	Parameter error (the address of the return parameter packet cannot be used)

[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)

TALM_STP	The alarm handler is inactive
TALM_STA	The alarm handler is active

```
#define  TALM_STP  0x00 /* alarm handler is inactive    */
#define  TALM_STA  0x01 /* alarm handler is active      */
```

If the alarm handler is active (TALM_STA), lfttim returns the relative time until the alarm handler is scheduled to start. This value is within the range almtim ≥ lfttim ≥ 0 specified with tk_sta_alm. Since lfttim is decremented with each timer interrupt, lfttim = 0 means the alarm handler will start at the next timer interrupt. If the alarm handler is inactive (TALM_STP), lfttim is indeterminate. If the alarm handler specified with tk_ref_alm in almid does not exist, error code E_NOEXS is returned.

4.7 Interrupt Management Functions

The interrupt management function executes operations such as handler definition or interrupt control for external interrupts and CPU exceptions.

[Difference with T-Kernel]

There are no functions related to interrupt controller control. The interrupt controller control is largely hardware dependent, so it is determined not to be specified in µT-Kernel.

4.7.1 Interrupt Handler Management

An interrupt handler runs as a task-independent portion. System calls can be invoked in a task-independent portion in the same way as in a task portion, but the following restriction applies to system call issuing in a task-independent portion.

- A system call that implicitly specifies the invoking task, or one that may put the invoking task in WAIT state cannot be issued. Error code `E_CTX` is returned in such cases.

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.

tk_def_int

Define Interrupt Handler

[C Language Interface]

```
ER ercd = tk_def_int ( UINT dintno, T_DINT *pk_dint ) ;
```

[Parameters]

```
UINT        dintno      Interrupt definition number

T_DINT*     pk_dint     Packet of interrupt handler definition information
pk_dint detail:
    ATR     intatr      Interrupt handler attributes
    FP      inthdr      Interrupt handler address
```

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

[Return Parameters]

```
ER          ercd        Error code
```

[Error Codes]

E_OK	Normal completion
E_NOMEM	Insufficient memory (memory for control block cannot be allocated)
E_RSATR	Reserved attribute (intatr is invalid or cannot be used)
E_PAR	Parameter error (dintno, pk_dint, or inthdr is invalid or cannot be used)

[Description]

Defines an interrupt handler for interrupt definition number dintno, and enables the use of the interrupt handler. Here “interrupts” include both external interrupts from a device and CPU exceptions.

This system call maps the interrupt definition number indicated in dintno to the interrupt handler address and attributes.

The specific significance of dintno is defined separately for each implementation, but generally it means an interrupt vector number.

intatr indicates system attributes in its low bits, with the high bits used for implementation-dependent attributes. The system attributes part of intatr is specified in the following format.

intatr := (TA_ASM || TA_HLNG)

TA_ASM	The handler is written in assembly language
TA_HLNG	The handler is written in a high-level language

```
#define  TA_ASM   0x00000000  /* assembly program              */
#define  TA_HLNG  0x00000001  /* high-level language program   */
```

When the TA_ASM attribute is specified, in principle the OS is not involved in interrupt handler starting. When an interrupt occurs, the interrupt handler defined by this system call is directly invoked by the interrupt handling system of the CPU hardware. Accordingly, processing for saving and restoring registers used by the interrupt handler is necessary at the beginning and end of the interrupt handler. An interrupt handler is terminated by execution of the tk_ret_int system call or by the CPU interrupt return instruction (or equivalent means).

Provision of a means for return from an interrupt handler without using tk_ret_int and without OS intervention is mandatory. Note that if tk_ret_int is not used, delayed dispatching is not necessary.

Support for return from an interrupt handler using tk_ret_int is mandatory, and in this case delayed dispatching is necessary.

When the TA_HLNG attribute is specified, the interrupt handler is started via a high-level language support routine. The high-level language support routine takes care of saving and restoring register values. The interrupt handler terminates by a simple return from the function. The interrupt handler takes the following format when the TA_HLNG attribute is specified.

```
void inthdr( UINT dintno )
{
        /*
        Processing
        */
        return; /* exit interrupt handler */
}
```

The parameter dintno passed to an interrupt handler is a number identifying the interrupt that was raised, and is the same as that specified with tk_def_int. Depending on the implementation, other information about the interrupt may be passed in addition to dintno. If such information is used, it must be defined for each implementation in a second parameter or subsequent parameters passed to the interrupt handler.

If the TA_HLNG attribute is specified, it is assumed that the CPU interrupt flag will be set to interrupts disabled state from the time the interrupt is raised until the interrupt handler is called. In other words, as soon as an interrupt is raised, the state goes to multiple interrupts disabled, and this state remains when the interrupt handler is called. If multiple interrupts are enabled, the interrupt handler must include processing that enables interrupts by manipulating the CPU interrupt flag.

Also in the case of the TA_HLNG attribute, upon entry into the interrupt handler, system call issuing must be possible. Note, however, that assuming standard provision of the functionality described above, extensions, such as adding a function for entering an interrupt handler with multiple interrupts enabled, are allowed.

When the TA_ASM attribute is specified, the state upon entry into the interrupt handler is separately defined for each implementation. Issues such as the status of the stack and registers upon interrupt handler entry, whether system calls can be made, the method of invoking system calls, and the method of returning from the interrupt handler without OS intervention must all be defined explicitly.

In the case of the TA_ASM attribute, depending on the implementation, there may be cases where interrupt handler execution is not considered to be a task-independent portion. In such a case, the following points need to be noted carefully.

- If interrupts are enabled, there is a possibility that task dispatching will occur.
- When a system call is invoked, it will be processed as having been called from a task portion or quasi-task portion.

If a method is provided for performing some kind of operation in an interrupt handler to have it detected as task-independent portion, that method must be indicated for each implementation.

Even if a system call is invoked from an interrupt handler and this causes the task in RUN state up to that time to go to another state, with a different task going to RUN state, dispatching (task switching) does not occur while the interrupt handler is running. Completion of execution by the interrupt handler has precedence even if dispatching is necessary; dispatching takes place only when the interrupt handler terminates. In other words, a dispatch request occurring 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 in an interrupt handler, to call a system call that can enter WAIT state, or one that is intended for the invoking task.

If `pk_dint`=NULL, the predefined interrupt handler definition shall be cleared. If an interrupt occurs in the state of clearing the interrupt handler definition, the behavior is implementation-defined.

It is possible to redefine an interrupt handler for an interrupt number already having a defined handler. It is not necessary first to cancel the definition for that number. Defining a new handler for a `dintno` already having an interrupt handler defined does not return an error.

[Additional Notes]

The various specifications governing the `TA_ASM` attribute are mainly concerned with achieving an interrupt hook. For example, if exception occurs due to an access to an illegal address, usually the exception is detected by an interrupt handler that a program at an upper level defines, and then the error is processed. However, while debugging, a debugger is invoked instead of error processing by the program at an upper level. In this case, the interrupt handler defined by the program at the upper level hooks the interrupt handler for the debugger. Depending on the situation, the handler passes the exception to the interrupt handling routine for the debugger or just processes the exception by itself.

[Porting Guideline]

The specific meaning of `dintno` can vary depending upon implementation differences such as hardware or OS, so you should check the implementation specifications in full when porting.

When porting it to the T-Kernel, the function (DINTNO) of converting from the interrupt vector to the interrupt definition number may exist as a T-Kernel/SM function. Therefore, it is better to refer to this as well.

tk_ret_int

Return from Interrupt Handler

[C Language Interface]

```
void tk_ret_int ( ) ;
```

- Although this system call is defined in the form of a C language interface, it will not be called in this format if a high-level language support routine is used.

[Parameters]

None.

[Return Parameters]

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

[Error Codes]

- The following kind of error may be detected, but no return is made to the context issuing the system call even if the error is detected. For this reason, the error code cannot be passed directly as a system call return parameter. If an error is detected, the behavior is implementation-defined.

```
E_CTX  Context error (issued from other than an interrupt handler (implementation-dependent
       error))
```

[Description]

Exits 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 to end 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 OS is not involved in the starting of a `TA_ASM` attribute interrupt handler. When an interrupt is raised, the defined interrupt handler is started directly by the interrupt system of the CPU hardware. The saving and restoring of registers used by the interrupt handler must therefore be handled in the interrupt handler.

For the same reason, the stack and register states at the time `tk_ret_int` is issued must be the same as those at the time of entry into the interrupt handler. When using a trap instruction to implement this, the function code may be unavailable in `tk_ret_int`. In this case, `tk_ret_int` may be implemented using a trap instruction with a vector different from that used for other system calls.

[Additional Notes]

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

Using an assembly-language return (REIT) instruction instead of `tk_ret_int` to exit the interrupt handler is possible if it is clear no dispatching will take place on return from the handler (the same task is guaranteed to continue executing), or if there is no need for dispatching to take place.

Depending on the CPU architecture and method of configuring the OS, it may be possible to perform delayed dispatching even when an interrupt handler exits using an assembly-language REIT instruction. In such cases, the assembly-language REIT instruction may be interpreted as though it were a `tk_ret_int` system call.

When `tk_ret_int` is called from a time event handler, whether `E_CTX` error is checked or not is implementation-defined. Depending on the implementation, control may return from a different type of handler.

4.7.2 CPU Interrupt Control

These functions are for CPU external interrupt flag control. Generally they do not perform any operation on the interrupt controller.

The interrupt system is highly hardware dependent and varies according to systems, so it is difficult to standardize interrupt control. Therefore, with respect to the interrupt control, only DI, EI, and isDI are defined as the standard specification.

DI, EI, and isDI 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.

DI

```
DI( UINT intsts )
```

`intsts` CPU interrupt status (details are implementation-dependent)

This is not a pointer.

Disables all external interrupts.

The status prior to disabling interrupts is stored in `intsts`.

EI

```
EI( UINT intsts )
```

`intsts` CPU interrupt status (details are implementation-dependent)

Enables all external interrupts.

More precisely, this macro restores the status in `intsts`. That is, the interrupt status reverts to what it was before interrupts were disabled by DI(). If there were interrupts disabled at the time DI() was executed, those interrupts are not enabled by EI(). All interrupts can be enabled, however, by specifying 0 in `intsts`.

`intsts` must be either the values stored in it by DI() or 0. If any other value is specified, the behavior is not guaranteed.

isDI

```
BOOL isDI( UINT intsts )
```

`intsts` CPU interrupt status (details are implementation-dependent)

Return code:

TRUE (not 0):	Interrupts disabled
FALSE:	Interrupts enabled

Gets the status of external interrupt disabling stored in `intsts`.

Interrupts disabled status is the status in which µT-Kernel determines that interrupts are disabled.

`intsts` must be the value stored by DI(). If any other value is specified, the behavior is not guaranteed.

Sample usage:

```
void foo()
```

```
{
         UINT intsts;
         DI(intsts);
         if ( isDI(intsts) ) {
            /* Interrupts were already disabled at the time this function was called.*/
         } else {
            /* Interrupts were enabled at the time this function was called. */
         }
     EI(intsts);
}
```

4.8 System Management Functions

System management functions are functions for changing and referencing system states. Functions are provided for rotating task precedence in a queue, getting the ID of the task in RUN state, disabling and enabling task dispatching, referencing context and system states, and referencing the µT-Kernel version.

[Difference with T-Kernel]

There is no function (`tk_set_pow`) to set low-power mode.

tk_rot_rdq

Rotate Ready Queue

[C Language Interface]

```
ER ercd = tk_rot_rdq ( PRI tskpri ) ;
```

[Parameters]

PRI	tskpri	Task priority

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_PAR	Parameter error (tskpri is invalid)

[Description]

Rotates the precedence among tasks having the priority specified in tskpri.

This system call changes the precedence of tasks in RUN or READY state having the specified priority so that the task with the highest precedence among those tasks is given the lowest precedence.

By setting tskpri = TPRI_RUN = 0, this system call rotates the precedence of tasks having the priority level of the task currently in RUN 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 either TPRI_RUN or the current priority of the invoking task as the priority, the precedence of the invoking task will become the lowest among tasks of the same priority. In this way, the system call can be used to relinquish execution privilege.

In dispatch disabled state, the task with highest precedence among tasks of the same priority is not always the currently executing task. The precedence of the invoking task will therefore not always become the lowest among tasks having the same priority when the above method is used in dispatch disabled state.

Examples of tk_rot_rdq execution are given in Figures 4.7 and 4.8. When this system call is issued in the state shown in Figure 4.7 specifying tskpri = 2, the new precedence order becomes that in Figure 4.8, and Task C becomes the executing task.

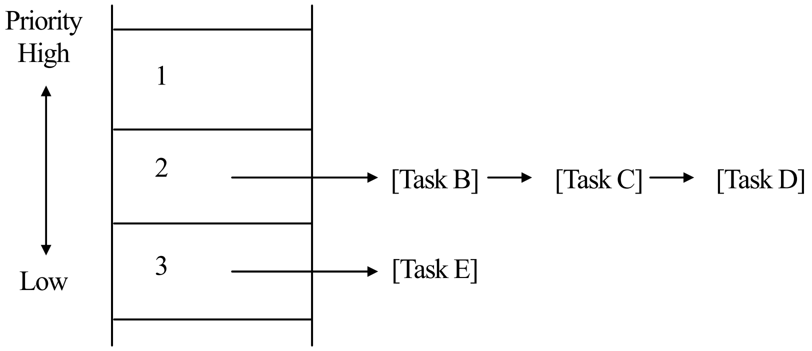
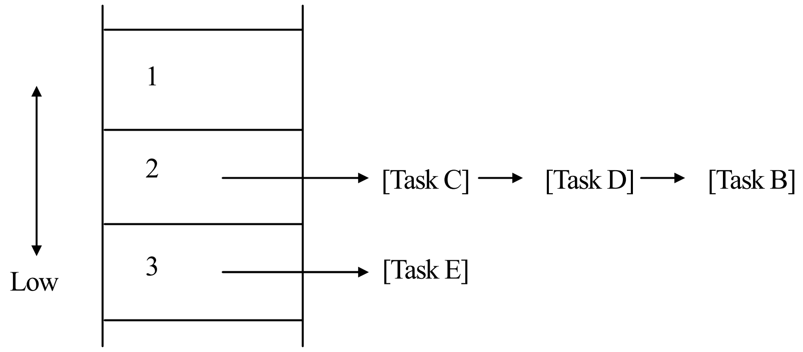


Figure 4.7: Precedence Before Issuing tk_rot_rdq



- Task C executes next.

Figure 4.8: Precedence After Issuing tk_rot_rdq (tskpri = 2)

tk_get_tid

Get Task Identifier

[C Language Interface]

```
ID tskid = tk_get_tid ( ) ;
```

[Parameters]

None

[Return Parameters]

```
ID          tskid       ID of the task in RUN state
```

[Error Codes]

None

[Description]

Gets the ID number of the task currently in RUN state. Unless the task-independent portion is executing, the current RUN state task will be the invoking task.

If there is no task currently in RUN state, 0 is returned.

[Additional Notes]

The task ID returned by `tk_get_tid` is identical to `runtskid` returned by `tk_ref_sys`.

tk_dis_dsp

Disable Dispatch

[C Language Interface]

```
ER ercd = tk_dis_dsp ( ) ;
```

[Parameters]

None

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_CTX	Context error (issued from task-independent portion)

[Description]

Disables task dispatching. Dispatch disabled state remains in effect until tk_ena_dsp is called to enable task dispatching. While dispatching is disabled, the invoking task does not change from RUN state to READY state or to WAIT state. External interrupts, however, are still enabled, so even in dispatch disabled state an interrupt handler can be started. In dispatch disabled state, the running task can be preempted by an interrupt handler, but not by another task.

The specific operations during dispatch disabled state are as follows.

- Even if a system call issued from an interrupt handler or by the task that called tk_dis_dsp results in a task going to READY state with a higher priority than the task that called tk_dis_dsp, that task will not be dispatched. 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 WAIT state (e.g., tk_slp_tsk or tk_wai_sem), error code E_CTX is returned to it.
- When system status is referenced by tk_ref_sys, TSS_DDSP is returned in sysstat.

If tk_dis_dsp is called for a task already in dispatch disabled state, that state continues with no error code returned. No matter how many times tk_dis_dsp is called, calling tk_ena_dsp just once is sufficient to enable dispatching again. The operation when the pair of system calls tk_dis_dsp and tk_ena_dsp are nested must therefore be managed by the user as required.

[Additional Notes]

A task in RUN state cannot go to DORMANT state or NON-EXISTENT state while dispatching is disabled. If tk_ext_tsk or tk_exd_tsk is called for a task in RUN state while interrupts or dispatching is disabled, error code E_CTX is detected. Since, however, tk_ext_tsk and tk_exd_tsk are system calls that do not return to their original context, such errors are not passed in return parameters by these system calls.

Use of dispatch disabled state for mutual exclusion control among tasks is possible only if the system does not have a multiprocessor configuration.

tk_ena_dsp

Enable Dispatch

[C Language Interface]

```
ER ercd = tk_ena_dsp ( ) ;
```

[Parameters]

None

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_CTX	Context error (issued from task-independent portion)

[Description]

Enables task dispatching.

This system call cancels the disabling of dispatching by the `tk_dis_dsp` system call.

If `tk_ena_dsp` is called for a task not in dispatch disabled state, the dispatch enabled state continues and no error code is returned.

tk_ref_sys

Reference System Status

[C Language Interface]

```
ER ercd     = tk_ref_sys ( T_RSYS *pk_rsys ) ;
```

[Parameters]

T_RSYS*	pk_rsys	Address of packet for returning status information

[Return Parameters]

ER	ercd	Error code

pk_rsys detail:

UINT	sysstat	System status
ID	runtskid	ID of task currently in RUN state
ID	schedtskid	ID of task scheduled to run next

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

[Error Codes]

E_OK	Normal completion
E_PAR	Parameter error (the address of the return parameter packet cannot be used)

[Description]

Gets the current system execution status, passing in return parameters system information such as the dispatch disabled state and whether a task-independent portion is executing.

The following values are returned in sysstat.

```
sysstat  := ( TSS_TSK | [TSS_DDSP] | [TSS_DINT] )
          || ( TSS_QTSK | [TSS_DDSP] | [TSS_DINT] )
          || ( TSS_INDP )
```

TSS_TSK	Task portion executing
TSS_DDSP	Dispatch disabled
TSS_DINT	Interrupts disabled
TSS_INDP	Task-independent portion executing
TSS_QTSK	Quasi-task portion executing

```
#define  TSS_TSK   0   /* Task portion executing                */
#define  TSS_DDSP  1   /* Dispatch disabled                     */
#define  TSS_DINT  2   /* Interrupts disabled                   */
#define  TSS_INDP  4   /* Task-independent portion executing    */
#define  TSS_QTSK  8   /* Quasi-task portion executing          */
```

The ID of the task currently in RUN state is returned in `runtskid`, while `schedtskid` indicates the ID of the next task scheduled to go to RUN state. Normally `runtskid` = `schedtskid`, but this is not necessarily true if, for example, a higher-priority task was wakened during dispatch disabled state. If there is no such task, 0 is returned.

It must be possible to invoke this system call from an interrupt handler or time event handler.

[Additional Notes]

When `tk_ref_sys` is called in the task or handler of `TA_ASM` attribute, correct information is not necessarily returned.

[Difference with T-Kernel 1.00.00]

The type of `sysstat` is UINT instead of INT.

tk_ref_ver

Reference Version Information

[C Language Interface]

```
ER ercd = tk_ref_ver ( T_RVER *pk_rver ) ;
```

[Parameters]

```
T_RVER*     pk_rver     Start address of packet for version information
```

[Return Parameters]

```
ER          ercd        Error code
```

```
pk_rver detail:
    UH      maker       μT-Kernel maker code
    UH      prid        μT-Kernel ID
    UH      spver       Specification version
    UH      prver       μT-Kernel version
    UH      prno[4]     μT-Kernel products management information
```

[Error Codes]

`E_OK`	Normal completion
`E_PAR`	Parameter error (the address of the return parameter packet cannot be used)

[Description]

Gets information about the μT-Kernel version in use, returning that information in the packet specified in `pk_rver`. The following information can be obtained.

`maker` is the vendor code of the μT-Kernel implementing vendor. The `maker` field has the format shown in Figure 4.9.

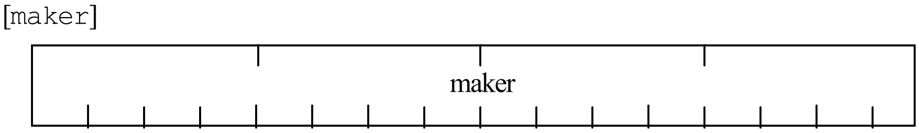


Figure 4.9: maker Field Format

`prid` is a number indicating the μT-Kernel type. The `prid` format is shown in Figure 4.10.

Assignment of values to `prid` is left up to the vendor implementing the μT-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 μT-Kernel type.

[prid]

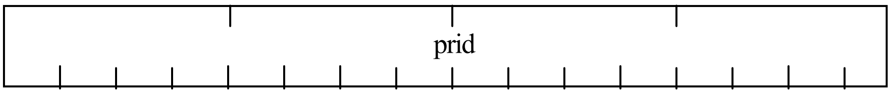


Figure 4.10: prid Field Format

The upper 4 bits of `spver` give the TRON specification series. The low 12 bits indicate the µT-Kernel specification version implemented. The format of `spver` is shown in Figure 4.11.

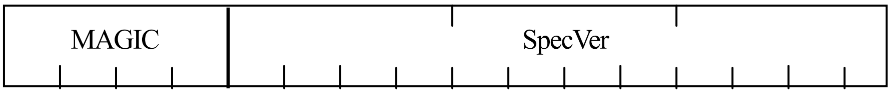
If, for example, a product conforms to the µT-Kernel specification Ver 1.02.xx, `spver` is as follows.

```
MAGIC       = 0x6       (µT-Kernel)
SpecVerS    = 0x102     (Ver 1.02)
Spver       = 0x6102
```

If a product implements the µT-Kernel specification draft version Ver 1.B0.xx, `spver` is as follows.

```
MAGIC       = 0x6       (µT-Kernel)
SpecVerS    = 0x1B0     (Ver 1.B0)
Spver       = 0x61B0
```

[spver]



MAGIC: number for distinguishing OS series

0x0	common to TRON (TAD, etc.)
0x1	reserved
0x2	reserved
0x3	reserved
0x4	reserved
0x5	reserved
0x6	µT-Kernel
0x7	T-Kernel

SpecVer: The version of the TRON specification on which the product is based. This is given as a three-digit packed-format BCD code. In the case of a draft version, the letter A, B, or C may appear in the second digit. In this case, the corresponding hexadecimal form of A, B, or C is inserted.

Figure 4.11: spver Field Format

`prver` is the version number of the µT-Kernel implementation. The specific values assigned to `prver` are left to the vendor implementing the µT-Kernel to decide. `prno` is a return parameter for use in indicating µT-Kernel product management information, product number, etc. The specific meaning of values set in `prno` is left to the vendor implementing the µT-Kernel to decide.

[Additional Notes]

The format of the packet and structure members for getting version information is mostly uniform across the various TRON specifications, but the CPU information and variation descriptors are not specified. The value obtained by `tk_ref_ver` in `SpecVer` is the first three digits of the specification version number. The numbers after that indicate minor revisions such as those issued to correct misprints and the like, and are not obtained by `tk_ref_ver`. For the purpose of matching to the specification contents, the first three numbers of the specification version are sufficient.

An OS implementing a draft version may have A, B, or C as the second number of `SpecVer`. It must be noted that in such cases the specification order of release may not correspond exactly to higher and lower `SpecVer` values. For example, specifications may be released in the following order:

Ver 1.A1 → Ver 1.A2 → Ver 1.B1 → Ver 1.C1 → Ver 1.00 → Ver 1.01 → …

In this example, when going from Ver 1.Cx to Ver 1.00, `SpecVer` goes from a higher to a lower value.

[Difference with T-Kernel 1.00.00]

The assignment of MAGIC for `spver`, the member of `T_RVER`, is different. This is because the policy and the naming scheme of system call, etc. are different between the OS specifications in previous TRON systems and those in a T-Kernel system, so that it is determined not to be managed in the same system.

4.9 Subsystem Management Functions

The subsystems in µT-Kernel are only composed of the extended SVC handlers for receiving requests from an application, etc.

[Difference with T-Kernel]

The following system calls do not exist:

- `tk_sta_ssy` calls the startup function
- `tk_cln_ssy` calls the cleanup function
- `tk_evt_ssy` calls the event processing function
- `tk_cre_res` creates a resource group
- `tk_del_res` deletes a resource group
- `tk_get_res` gets the resource management block

tk_def_ssy

Define Subsystem

[C Language Interface]

```
ER ercd = tk_def_ssy ( ID ssid, T_DSSY *pk_dssy ) ;
```

[Parameters]

```
ID          ssid        Subsystem ID
T_DSSY*     pk_dssy     Subsystem definition information
pk_dssy detail:
    ATR     ssyatr      Subsystem attributes
    PRI     ssypri      Subsystem priority
    FP      svchdr      Extended SVC handler address
    FP      breakfn     Break function address
    FP      startupfn   Startup function address
    FP      cleanupfn   Cleanup function address
    FP      eventfn     Event handling function address
    W       resblksz    Resource control block size (in bytes)
```

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

[Return Parameters]

```
ER          ercd        Error code
```

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (ssid is invalid or cannot be used)
E_NOMEM	Insufficient memory (memory for control block cannot be allocated)
E_RSATR	Reserved attribute (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)

[Description]

Defines subsystem ssid.

A subsystem ID must be assigned to each subsystem without overlapping with other subsystems. The OS does not have a function for assigning these automatically.

Subsystem IDs 1 to 9 are reserved for µT-Kernel use. 10 to 255 are numbers used by middleware, etc. The maximum usable subsystem ID value is implementation-dependent and may be lower than 255 in some implementations.

ssyatr indicates system attributes in its low bits and implementation-dependent attributes in the high bits. The system attributes in ssyatr are not assigned in this version, and no system attributes are used.

ssypri has the priority of the subsystem, but this member is retained only to ensure compatibility with T-Kernel, so this specification is meaningless in µT-Kernel.

NULL can be specified in breakfn, startupfn, cleanupfn, and eventfn, in which case the corresponding function will not be called.

The pointer to a function can be specified in breakfn, startupfn, cleanupfn, and eventfn, but these functions are never called from the OS. These members are retained for compatibility with T-Kernel.

- Resource control block

  µT-Kernel does not have resource management blocks, so whatever is specified for resblksz does not affect the OS's behavior. This member is retained for compatibility with T-Kernel.

- 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( VP pk_para, FN fncd )
{
      /*
      Branching by fncd
      */

      return retcode; /* exit extended SVC handler */
}
```

  fncd is a function code. The low 8 bits of the instruction code are the subsystem ID. The remaining high bits can be used in any way by the subsystem. Ordinarily, they are used as a function code inside the subsystem. A function code must be a positive value, so the most significant bit is always 0.

  pk_para points to a packet of parameters passed to this system call. The packet format can be decided by the subsystem. Generally a format like the stack passed to a C language function is used, which in many cases is the same format as a C language structure.

  The return code passed by an extended SVC handler is passed to the caller transparently as the function return value. As a rule, negative values are error codes and 0 or positive values are the return code for normal completion. If an extended SVC call fails for some reason, the OS error code (which is also a negative value) is returned to the caller without invoking the extended SVC handler, so it is best to avoid confusion with these values.

  The extended SVC should be specified so that it can be invoked in the C language function format without depending on the OS implementation. The subsystem must provide an interface library for converting from the C language function format to the OS-dependent extended SVC calling format.

  An extended SVC handler runs as a quasi-task portion. It can be called from a task-independent portion, and in this case the extended SVC handler also runs as a task-independent portion.

- break function, startup function, cleanup function, and event processing function

In µT-Kernel, these functions are never called.

[Difference with T-Kernel]

In µT-Kernel, the break function, startup function, cleanup function, and event function is never called by the OS. Also, there is no function related to resource management blocks. As long as they are used only in µT-Kernel, these specifications mean nothing (whatever is specified does not affect the behavior of the OS).

Subsystem priority is used only to order the calling of each startup function, cleanup function, and event processing function in the T-Kernel. These functions are not used in subsystems written for µT-Kernel, therefore, whatever is specified for its value does not affect the behavior of the OS. However, in terms of T-Kernel portability, it is desirable to specify a value in the range of 1-16 as required in the specification.

[Difference with T-Kernel 1.00.00]

`resblksz`, the member of `T_DSSY`, is of W type instead of INT type.

[Porting Guideline]

To make it possible to port an extended SVC created for µT-Kernel without any changes, NULL should be specified for `breakfn`, `startupfn`, `cleanupfn`, and `eventfn`. Also, although less relevant for portability, in order not to allocate useless memory when ported to T-Kernel, it is desirable to specify zero for `resblksz`. Thereafter, you can port the extended the SVC for µT-Kernel into the T-Kernel simply by recompiling it.

Note that in an environment of 16-bit width, only 7 bits (0-127) of INT type is available as a function code.

tk_ref_ssy

Reference Subsystem Status

[C Language Interface]

```
ER ercd = tk_ref_ssy ( ID ssid, T_RSSY *pk_rssy ) ;
```

[Parameters]

ID	ssid	Subsystem ID
T_RSSY*	pk_rssy	Subsystem definition information

[Return Parameters]

ER	ercd	Error code

pk_rssy detail:

PRI	ssypri	Subsystem priority
W	resblksz	Resource control block size (in bytes)

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

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (ssid is invalid or cannot be used)
E_NOEXS	Object does not exist (the subsystem specified in ssid is not defined)
E_PAR	Parameter error (pk_rssy is invalid or cannot be used)

[Description]

References information about the status of the subsystem specified in ssid. ssypri and resblksz are retained as members of the returned packet to ensure compatibility with T-Kernel. They are not used in µT-Kernel, so the values set to these are not particularly defined and shall be implementation-defined. If the subsystem specified in ssid does not exist, E_NOEXS is returned.

[Difference with T-Kernel 1.00.00]

resblksz, the member of T_RSSY, is of W type instead of INT type.

4.10 Device Management Functions

4.10.1 Basic Concepts

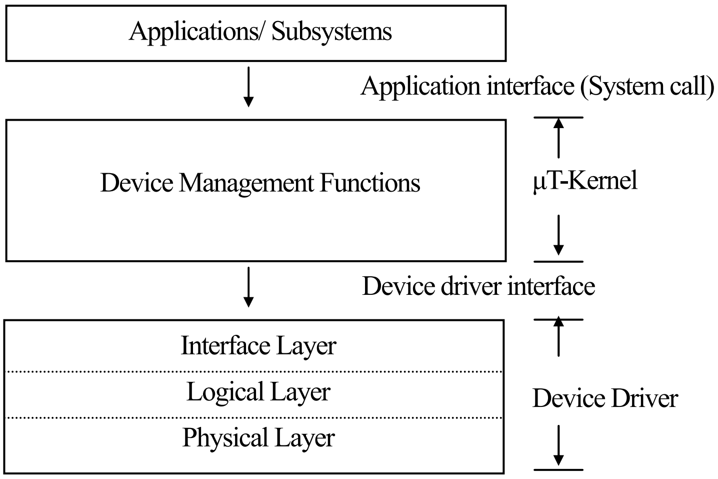


Figure 4.12: Device Management Functions

(1) Device Name (UB* type)

A device name is a string of up to 8 characters consisting of the following elements.

```
#define  L_DEVNM 8   /* Device name length */
```

Type Name indicating the device type

Characters a to z and A to Z can be used.

Unit One letter indicating a physical device

Each unit is assigned a letter from a to z in order starting from a.

Subunit One to three digits indicating a logical device

Each subunit is assigned a number from 0 to 254 in order starting from 0.

Device names take the format type + unit + subunit. Some devices may not have a unit or subunit, in which case the corresponding field is omitted.

A name consisting of type + unit is called a physical device name. A name consisting of type + unit + subunit may be called a logical device name to distinguish it from a physical device name. If there is no subunit, the physical device name and logical device name are identical. The term "device name" by itself means the logical device name.

A subunit generally refers to a partition on a hard disk but can be used to mean other logical devices as well.

Examples:

hda	Hard disk (entire disk)
hda0	Hard disk (1st partition)
fda	Floppy disk
rsa	Serial port

kbpd	Keyboard/pointing device

(2) Device ID (ID type)

By registering a device (device driver) with μT-Kernel, a device ID (> 0) is assigned to the device (physical device name). Device IDs are assigned to each physical device. The device ID of a logical device consists of the device ID assigned to the physical device to which is appended the subunit number + 1 (1 to 255).

devid: The device ID assigned at device registration

devid	Physical device
devid + n + 1	The nth subunit (logical device)

Examples:

hda	devid	Entire hard disk
hda0	devid + 1	1st partition of hard disk
hda1	devid + 2	2nd partition of hard disk

(3) Device Attribute (ATR type)

Device attributes are defined as follows, in order to classify devices by their properties.

```
IIII IIII IIII IIII PRxx xxxx KKKK KKKK
```

The upper 16 bits are device-dependent attributes defined for each device.

The lower 16 bits are standard attributes defined as follows.

```
#define  TD_PROTECT      0x8000  /* P: write protection             */
#define  TD_REMOVABLE    0x4000  /* R: removable media              */

#define  TD_DEVKIND      0x00ff  /* K: device/media kind            */
#define  TD_DEVTYPE      0x00f0  /* device type                     */

                                 /* device type                     */
#define  TDK_UNDEF       0x0000  /* undefined/unknown               */
#define  TDK_DISK        0x0010  /* disk device                     */

                                 /* disk kind                       */
#define  TDK_DISK_UNDEF  0x0010  /* miscellaneous disk              */
#define  TDK_DISK_RAM    0x0011  /* RAM disk (used as main memory)  */
#define  TDK_DISK_ROM    0x0012  /* ROM disk (used as main memory)  */
#define  TDK_DISK_FLA    0x0013  /* Flash ROM or other silicon disk */
#define  TDK_DISK_FD     0x0014  /* floppy disk                     */
#define  TDK_DISK_HD     0x0015  /* hard disk                       */
#define  TDK_DISK_CDROM  0x0016  /* CD-ROM                          */
```

Currently, no device types other than disks are defined. Other devices are assigned to undefined type (TDK_UNDEF). Note that device types are defined for the sake of distinguishing devices from the standpoint of the user as necessary, such as when applications must change their processing based on the type of device or media. Devices for which no such distinctions are necessary do not have to have a device type assigned. See the individual device driver specifications regarding device-dependent attributes.

(4) Device Descriptor (ID type)

A device descriptor (> 0) is an identifier used for accessing a device, assigned by μT-Kernel when a device is opened.

A device descriptor belongs to a resource group. Operations using a device descriptor can be performed only by tasks belonging to the same resource group as the device descriptor. Error code (E_OACV) is returned for requests from tasks belonging to a different resource group.

(5) Request ID (ID type)

When an IO request is made to a device, a request ID (> 0) is assigned identifying the request. This ID can be used to wait for IO completion.

(6) Data Number (W type)

Device data is specified by a data number. Data is classified into device-specific data and attribute data as follows.

Device-specific data: Data number ≥ 0

As device-specific data, the data numbers are defined separately for each device.

Examples

Disk	Data number = physical block number
Serial port	Data number = 0 only

Attribute data: Data number < 0

Attribute data specifies driver or device state acquisition and setting modes, and special functions, etc.

Data numbers common to devices are defined but device-dependent attribute data can also be defined. Details are given later.

[Difference with T-Kernel 1.00.00]

The type of data number is W instead of INT.

4.10.2 Application Interface

The functions below are provided as application interface functions, called as extended SVC. These functions cannot be called from a task-independent portion or while dispatch or interrupts are disabled (E_CTX).

```
ID   tk_opn_dev( UB *devnm, UINT omode )
ER   tk_cls_dev( ID dd, UINT option )
ID   tk_rea_dev( ID dd, W start, VP buf, W size, TMO tmout )
ER   tk_srea_dev( ID dd, W start, VP buf, W size, W *asize )
ID   tk_wri_dev( ID dd, W start, VP buf, W size, TMO tmout )
ER   tk_swri_dev( ID dd, W start, VP buf, W size, W *asize )
ID   tk_wai_dev( ID dd, ID reqid, W *asize, ER *ioer, TMO tmout )
INT  tk_sus_dev( UINT mode )
INT  tk_get_dev( ID devid, UB *devnm )
ID   tk_ref_dev( UB *devnm, T_RDEV *pk_rdev )
ID   tk_oref_dev( ID dd, T_RDEV *pk_rdev )
INT  tk_lst_dev( T_LDEV *pk_ldev, INT start, INT ndev )
INT  tk_evt_dev( ID devid, INT evttyp, VP evtinf )
```

tk_opn_dev

Open Device

[C Language Interface]

```
ID devid = tk_opn_dev ( UB *devnm, UINT omode ) ;
```

[Parameters]

UB*	devnm	Device name
UINT	omode	Open mode

[Return Parameters]

ID	devid	Device descriptor
	or	Error code

[Error Codes]

E_BUSY	Device busy (exclusive open)
E_NOEXS	Device does not exist
E_LIMIT	Open count exceeds the limit
Other	Errors returned by device driver

[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]

```
#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                     */
```

TD_READ Read only

TD_WRITE Write only

TD_UPDATE Sets read and write access mode.

When TD_READ is set, tk_wri_dev() cannot be used.

When TD_WRITE is set, tk_rea_dev() cannot be used.

TD_EXCL Exclusive

TD_WEXCL Exclusive write

TD_REXCL Exclusive read

Sets the exclusive mode.

When `TD_EXCL` is set, all concurrent opening is prohibited.

When `TD_WEXCL` is set, concurrent opening in write mode (`TD_WRITE` or `TD_UPDATE`) is prohibited.

When `TD_REXCL` is set, concurrent opening in read mode (`TD_READ` or `TD_UPDATE`) is prohibited.

Present Open Mode		Concurrent Open Mode							
		No exclusive mode		`TD_WEXCL`		`TD_REXCL`		`TD_EXCL`	
		R	W	R	W	R	W	R	W
No exclusive mode	R	Yes	Yes	Yes	Yes	No	No	No	No
	W	Yes	Yes	No	No	Yes	Yes	No	No
`TD_WEXCL`	R	Yes	No	Yes	No	No	No	No	No
	W	Yes	No	No	No	Yes	No	No	No
`TD_REXCL`	R	No	Yes	No	Yes	No	Yes	No	No
	W	No	Yes	No	No	No	No	No	No
`TD_EXCL`	R	No	No	No	No	No	Yes	No	No
	W	No	No	No	No	No	No	No	No

R= `TD_READ` W= `TD_WRITE` or `TD_UPDATE`

Yes = Can be opened No = Cannot be opened (`E_BUSY`)

Table 4.4: Whether Concurrent Open of Same Device is Allowed or NOT

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.

[Difference with T-Kernel]

Unnecessary lock (TD_NOLOCK) does not exist in the attribute for ‘`omode`’. This is because there is no virtual memory and no concept of resident memory in µT-Kernel.

tk_cls_dev

Close Device

[C Language Interface]

```
ER ercd = tk_cls_dev ( ID dd, UINT option ) ;
```

[Parameters]

ID	dd	Device descriptor
UINT	option	Close option

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_ID	dd is invalid or not open
Other	Errors returned by device driver

[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.

tk_rea_dev

Read Device

[C Language Interface]

```
ID reqid = tk_rea_dev ( ID dd, W start, VP buf, W size, TMO tmout ) ;
```

[Parameters]

ID	dd	Device descriptor
W	start	Read start location (≥ 0: Device-specific data, < 0: Attribute data)
VP	buf	Buffer location for putting the read data
W	size	Read size
TMO	tmout	Request acceptance timeout (ms)

[Return Parameters]

ID	reqid	Request ID
	or	Error code

[Error Codes]

E_ID	dd is invalid or not open
E_OACV	Open mode is invalid (read not permitted)
E_LIMIT	Number of requests exceeds the limit
E_TMOUT	Busy processing other requests
E_ABORT	Processing aborted
Other	Errors returned by device driver

[Description]

Starts reading device-specific data or attribute data from the specified device. This function only starts the reading, 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 processing the start of reading differs with the device; return of control is not necessarily immediate.

In the case of device-specific data, the start and size units are specified for each device. With attribute data, start is the attribute data number and size is in bytes. The attribute data of the data number specified in start is read. Normally, size must be at least as large as the size of the attribute data to be read. Reading of multiple attribute data in one operation is not possible.

When size = 0 is specified, actual reading does not take place but the current size of data that can be read is checked.

Whether or not a new request can be accepted while a read or write operation is in progress depends on the device driver. If a new request cannot be accepted, the request is queued. The timeout for request waiting is set in tmout. The TMO_POL or TMO_FEVR attribute can be specified for tmout. Note that what times out is request acceptance. Once a request has been accepted, this function does not time out.

[Difference with T-Kernel 1.00.00]

The type of `start` and `size` is W instead of INT.

However, since these items are implementation-dependent, implementation for `start` and `size` may be in the INT data type. These arguments are target system-dependent. For example, upper bits may always be unused if the implementation in W type is specified for a 16-bit CPU. Only in such cases, implementation in INT type instead of W type is allowed for the purpose of speeding the execution and making the program smaller.

tk_srea_dev

Synchronous Read Device

[C Language Interface]

```
ER ercd = tk_srea_dev ( ID dd, W start, VP buf, W size, W* asize ) ;
```

[Parameters]

ID	dd	Device descriptor
W	start	Read start location (≥ 0: Device-specific data, < 0: Attribute data)
VP	buf	Buffer location for putting the read data
W	size	Read size

[Return Parameters]

ER	ercd	Error code
W*	asize	Size of data being read

[Error Codes]

E_ID	dd is invalid or not open
	reqid is invalid or not a request for dd
E_OBJ	Another task is already waiting for request reqid
E_NOEXS	No requests are being processed (only when reqid = 0)
E_OACV	Open mode is invalid (read not permitted)
E_LIMIT	Number of requests exceeds the limit
E_ABORT	Processing aborted
Other	Errors returned by device driver

[Description]

Synchronous read. This is equivalent to the following.

```
        ER tk_srea_dev( ID dd, W start, VP 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;
        }
```

[Difference with T-Kernel 1.00.00]

The type of `start` and `size` as well as the type of the target pointed by `asize` are W instead of INT.

However, since these items are implementation-dependent, implementation for `start` and `size` may be in the INT data type. These arguments are target system-dependent. For example, upper bits may always be unused if the implementation in W type is specified for a 16-bit CPU. Only in such cases, implementation in INT type instead of W type is allowed for the purpose of speeding the execution and making the program smaller.

tk_wri_dev

Write Device

[C Language Interface]

```
ID reqid = tk_wri_dev ( ID dd, W start, VP buf, W size, TMO tmout ) ;
```

[Parameters]

ID	dd	Device descriptor
W	start	Write start location (≥ 0: Device-specific data, < 0: Attribute data)
VP	buf	Buffer holding data to be written
W	size	Size of data to be written
TMO	tmout	Request acceptance timeout (ms)

[Return Parameters]

ID	reqid	Request ID
	or	Error code

[Error Codes]

E_ID	dd is invalid or not open
E_OACV	Open mode is invalid (write not permitted)
E_RONLY	Read-only device
E_LIMIT	Number of requests exceeds the limit
E_TMOUT	Busy processing other requests
E_ABORT	Processing aborted
Other	Errors returned by device driver

[Description]

Starts writing device-specific data or attribute data to a device. This function only starts the writing, 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 processing the start of the write operation differs with the device; return of control is not necessarily immediate.

In the case of device-specific data, the start and size units are specified for each device. With attribute data, start is an attribute data number and size is in bytes. The attribute data of the data number specified in start is written. Normally size must be at least as large as the size of the attribute data to be written. Multiple attribute data cannot be written in one operation.

When size = 0 is specified, actual writing does not take place but the current size of data that can be written is checked.

Whether or not a new request can be accepted while a read or write operation is in progress depends on the device driver. If a new request cannot be accepted, the request is queued. The timeout for request waiting is set in tmout. The TMO_POL or TMO_FEVR attribute can be specified for tmout. Note that what times out is request acceptance. Once a request has been accepted, this function does not time out.

[Difference with T-Kernel 1.00.00]

The type of start and size is W instead of INT.

However, since these items are implementation-dependent, implementation for start and size may be in the INT data type. These arguments are target system-dependent. For example, upper bits may always be unused if the implementation in W type is specified for a 16-bit CPU. Only in such cases, implementation in INT type instead of W type is allowed for the purpose of speeding the execution and making the program smaller.

tk_swri_dev

Synchronous Write Device

[C Language Interface]

```
ER ercd = tk_swri_dev ( ID dd, W start, VP buf, W size, W* asize ) ;
```

[Parameters]

ID	dd	Device descriptor
W	start	Write start location (≥ 0: Device-specific data, < 0: Attribute data)
VP	buf	Buffer holding data to be written
W	size	Size of data to be written

[Return Parameters]

ER	ercd	Error code
W*	asize	Size of data being written

[Error Codes]

E_ID	dd is invalid or not open
	reqid is invalid or not a request for dd
E_OBJ	Another task is already waiting for request reqid
E_NOEXS	No requests are being processed (only when reqid = 0)
E_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	Errors returned by device driver

[Description]

Synchronous write. This is equivalent to the following.

```
ER tk_swri_dev( ID dd, W start, VP 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;
}
```

[Difference with T-Kernel 1.00.00]

The type of `start` and `size` is W instead of INT.

However, since these items are implementation-dependent, implementation for `start` and `size` may be in the INT data type. These arguments are target system-dependent. For example, upper bits may always be unused if the implementation in W type is specified for a 16-bit CPU. Only in such cases, implementation in INT type instead of W type is allowed for the purpose of speeding the execution and making the program smaller.

tk_wai_dev

Wait Device

[C Language Interface]

```
ID reqid = tk_wai_dev ( ID dd, ID reqid, W *asize, ER *ioer, TMO tmout ) ;
```

[Parameters]

ID	dd	Device descriptor
ID	reqid	Request ID
W*	asize	Returns the read/write data size
ER*	ioer	Returns IO error
TMO	tmout	Timeout (ms)

[Return Parameters]

ID	reqid	Completed request ID
	or	Error code

[Error Codes]

E_ID	dd is invalid or not open
	reqid is invalid or not a request for dd
E_OBJ	Another task is already waiting for request reqid
E_NOEXS	No requests are being processed (only when reqid = 0)
E_TMOUT	Timeout (processing continues)
E_ABORT	Processing aborted
Other	Errors returned by device driver

[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 requested processing results in error (IO error, etc.) ioer is stored rather than a return code. The return code is used for errors when the request wait itself was not handled properly. When error is passed in the return code, ioer has no meaning. Note also that if error is passed in the return code, tk_wai_dev must be called again to wait for completion, since the request processing is still ongoing.

If a task exception is raised during completion while waiting for completion using tk_wai_dev, the request in reqid is aborted and processing is completed. The result of aborting the requested processing is dependent on the device driver. When reqid = 0 was set, however, requests are not aborted but are treated as timeout. In this case, E_ABORT rather than E_TMOUT is returned.

It is not possible for multiple tasks to wait for completion of the same request ID at the same time. If there is a task waiting for request completion with reqid = 0 set, another task cannot wait for completion for the same device descriptor. Similarly, if there is a task waiting for request completion with reqid > 0 set, another task cannot wait for completion specifying reqid = 0.

[Difference with T-Kernel 1.00.00]

The type of the target pointed by asize is W instead of INT.

However, since this item is implementation-dependent, implementation for the type of the target pointed by asize may be in the INT data type. This argument is target system-dependent. For example, upper bits may always be unused if the implementation in W type is specified for a 16-bit CPU. Only in such cases, implementation in INT type instead of W type is allowed for the purpose of speeding the execution and making the program smaller.

tk_sus_dev

Suspend Device

[C Language Interface]

```
INT cnt = tk_sus_dev ( UINT mode ) ;
```

[Parameters]

UINT	mode	Mode

[Return Parameters]

INT	cnt	Suspend disable request count
	or	Error code

[Error Codes]

E_BUSY	Suspend already disabled
E_QOVR	Suspend disable request count limit exceeded

[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 /* forcibly suspend                 */
```

TD_SUSPEND Suspend

If suspending is enabled, suspends processing. If suspending is disabled, returns E_BUSY.

TD_SUSPEND | TD FORCE Forcibly suspend

Suspends even in suspend disabled state.

TD_DISSUS Disable suspension

Disables suspension.

TD_ENASUS Enable suspension

Enables suspension.

If the enable request count is greater than 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.Suspension processing in non-disk devices

2.Suspension processing in disk devices

3.SUSPEND state entered

Resumption from SUSPEND state is performed in the following steps.

1.Return from SUSPEND state

2.Resumption processing in disk devices

3.Resumption processing in non-disk devices

The number of suspend disable requests is counted. Suspension is enabled only if the same number of suspend enable requests are 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 upper limit of suspend-disabled request count is implementation-defined, but the count can be executed up to at least 255. When the upper limit is exceeded, `E_QOVR` is returned.

tk_get_dev

Get Device Name

[C Language Interface]

```
ID phyid = tk_get_dev ( ID devid, UB *devnm ) ;
```

[Parameters]

ID	devid	Device ID
UB*	devnm	Device name storage location

[Return Parameters]

ID	phyid	Device ID of physical device
	or	Error code

[Error Codes]

E_NOEXS	The device specified in devid does not exist

[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 LDEVNM + 1 bytes or larger.

The device ID of the physical device to which device devid belongs is passed in the return code.

tk_ref_dev
tk_oref_dev

Reference Device
Reference Device

[C Language Interface]

```
ID devid = tk_ref_dev ( UB *devnm, T_RDEV *pk_rdev ) ;
ID devid = tk_oref_dev ( ID dd, T_RDEV *pk_rdev ) ;
```

[Parameters]

UB*	devnm	Device name
ID	dd	Device descriptor
T_RDEV*	pk_rdev	Device information

[Return Parameters]

ID	devid	Device ID
	or	Error code

```
   pk_rdev detail:
        ATR devatr Device attributes
        W blksz    Block size of device-specific data (-1: unknown)
        INT nsub   subunit count
        INT subno   0: physical device: 1 to nsub: subunit number+1
        (Implementation-dependent information may be added beyond this point.)
```

[Error Codes]

E_NOEXS	The device specified in devnm does not exist.

[Description]

Gets information about the device specified in devnm or dd and puts the result in pk_rdev.

If pk_rdev = NULL is set, the device information is not stored. nsub indicates the number of physical device subunits belonging to the device specified in devnm or dd.

The device ID of the device specified in devnm is passed in the return code.

[Difference with T-Kernel 1.00.00]

blksz, the member of T_RDEV, is of W type instead of INT type.

tk_lst_dev

List Device

[C Language Interface]

```
INT cnt = tk_lst_dev ( T_LDEV *pk_ldev, INT start, INT ndev ) ;
```

[Parameters]

T_LDEV*	pk_ldev	Location of registered device information (array)
INT	start	Starting number
INT	ndev	Number to acquire

[Return Parameters]

INT	cnt	Remaining device registration count
	or	Error code

```
pk_ldev detail:
      ATR devatr        Device attributes
      W blksz           Block size of device-specific data (-1: unknown)
      INT   nsub        subunits
      UB devnm[L_DEVNM] physical device name
      (Implementation-dependent information may be added beyond this point.)
```

[Error Codes]

E_NOEXS	Start exceeds the number of registered devices

[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 pk_`ldev`. The space specified in pk_`ldev` must be large enough to hold `ndev` items of 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 returned. A value passed in return code less than or equal to `ndev` means all remaining registrations were obtained. Note that this numbering changes as devices are registered and deleted. For this reason, accurate information may not always be obtained if this information is acquired over multiple operations.

tk_evt_dev

Event Device

[C Language Interface]

```
INT retval = tk_evt_dev ( ID devid, INT evttyp, VP evtinf ) ;
```

[Parameters]

ID	devid	Event destination device ID
INT	evttyp	Driver request event type
VP	evtinf	Information for each event type

[Return Parameters]

INT	retval	Return code from device driver
	or	Error code

[Error Codes]

E_NOEXS	The device specified in devid does not exist
E_PAR	Internal device manager events (evttyp < 0) cannot be specified

[Description]

Sends a driver request event to the device (device driver) specified in devid. The following driver request events are defined.

```
#define  TDV_CARDEVT   1  /* PC Card event (see Card Manager)    */
#define  TDV_USBEVT    2  /* USB event (see USB Manager)         */
```

The functioning of driver request events and the contents of evtinf are defined for each event type.

4.10.3 Device Registration

The following device registration information is defined when registering a device.

Device registration is performed for each physical device.

```
typedef struct t_ddev {
   VP  exinf;   /* extended information                                */
   ATR drvatr;  /* driver attributes                                   */
   ATR devatr;  /* device attributes                                   */
   INT nsub;    /* subunits                                            */
   W   blksz;   /* block size of device-specific data (-1: unknown)    */
   FP  openfn;  /* open count                                          */
   FP  closefn; /* close count                                         */
   FP  execfn;  /* processing start function                           */
   FP  waitfn;  /* completion wait function                            */
   FP  abortfn; /* abort processing function                           */
   FP  eventfn; /* event function */
   /* Implementation-dependent information may be added beyond this point.
   */
} T_DDEV;
```

exinf is used to store any other desired information. The value of exinf is passed to the processing functions. Device management pays no attention to the contents. drvatr sets device driver attribute information. The low bits indicate system attributes and the high bits are used for implementation-dependent attributes. The implementation-dependent attribute portion is used, for example, to indicate validity flags when implementation-dependent data is added to T_DDEV.

```
drvatr := [TDA_OPENREQ]
#define  TDA_OPENREQ  0x0001  /* open/close each time */
```

TDA_OPENREQ

When a device is opened multiple times, normally openfn is called the first time it is opened and closefn the last time it is closed.

If TDA_OPENREQ is specified, then openfn/ closefn will be called for all open/ close operations even in case of multiple openings.

Device attributes are specified in devatr. The details of device attribute settings are as noted above. The number of subunits is set in nsub. If there are no subunits, 0 is specified.

blksz sets the block size of device-specific data in bytes. In the case of a disk device, this is the physical block size. It is set to 1 byte for a serial port, etc. For a device with no device-specific data it is set to 0. For an unformatted disk or other device whose block size is unknown, -1 is set.

If blksz ≤ 0, device-specific data cannot be accessed. When device-specific data is accessed by tk_rea_dev or tk_wri_dev, size × blksz must be the size of the area being accessed, that is, the size of buf.

openfn, closefn, execfn, waitfn, abortfn, and eventfn set the entry address of processing functions.

Details of the processing functions are discussed later.

[Difference with T-Kernel 1.00.00]

`blksz`, the member of `T_DDEV`, is of W type instead of INT type.

tk_def_dev

Define Device

[C Language Interface]

```
ID devid = tk_def_dev ( UB *devnm, T_DDEV *pk_ddev, T_IDEV *pk_idev ) ;
```

[Parameters]

UB	devnm	Physical device name
T_DDEV*	ddev	Device registration information
IDEV*	pk_idev	Returns device initialization information

[Return Parameters]

ID	devid	Device ID
	or	Error code

```
pk_idev detail:
ID        evtmbfid  Event notification message buffer ID
        (Implementation-dependent information may be added beyond this point.)
```

[Error Codes]

E_LIMIT	Number of registrations exceeds the system limit
E_NOEXS	The device specified in devnm does not exist (when pk_ddev = NULL)

[Description]

Registers a device with the device name set in devnm.

If a device with device name devnm is already registered, the registration is updated with new information in which case the device ID does not change.

When pk_ddev = NULL is specified, the registration of the device devnm is deleted.

The device initialization information is returned in pk_idev.

This includes information set by default when the device driver is started, and can be used as necessary.

When pk_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.

[Difference with T-Kernel]

When a device is registered or deregistered in T-Kernel, a notification is sent to each subsystem as follows: devid indicates the device ID of the physical device which was registered or deregistered. This function does not exist in the μ T-Kernel.

Device registration or update:	tk_evt_ssy(0, TSEVT_DEVICE_REGIST, 0, devid)
Device deletion:	tk_evt_ssy(0, TSEVT_DEVICE_DELETE, 0, devid)

tk_ref_idv

Refer Initial Device Information

[C Language Interface]

```
ER ercd = tk_ref_idv ( T_IDEV *pk_idev ) ;
```

[Parameters]

```
T_IDEV*     pk_idev     Returns device initialization information
```

[Return Parameters]

```
ER          ercd        Error code
```

```
pk_idev detail:
ID          evtmbfid  Event notification message buffer ID
       (Implementation-dependent information may be added beyond this point.)
```

[Error Codes]

E_MACV The area pointed to by pk_idev is invalid.

[Description]

Gets device initialization information.

The contents are the same as the information set by tk_def_dev().

4.10.4 Device Driver Interface

The device driver interface consists of processing functions specified when registering a device. These functions are called by device management and run as a quasi-task portion. They must be reentrant. The mutually exclusive calling of these processing functions is not guaranteed. If, for example, there are simultaneous requests from multiple devices for the same device, different tasks might call the same processing function at the same time. The device driver must apply mutual exclusion control in such cases as required.

IO requests to a device driver are made by means of the following request packet mapped to a request ID.

```
typedef struct t_devreq {
        struct     t_devreq *next;  /* I: Link to request packet (NULL: termination)     */
        VP         exinf;           /* X: Extended information                            */
        ID         devid;           /* I: Target device ID                               */
        INT        cmd:4;           /* I: Request command                                */
        BOOL       abort:1;         /* I: TRUE if abort request                          */
        W          start;           /* I: Starting data number                           */
        W          size;            /* I: Request size                                   */
        VP         buf;             /* I: IO buffer address                              */
        W          asize;           /* O: Size of result                                 */
        ER         error;           /* O: Error result                                   */
/* Implementation-dependent information may be added beyond this point.                  */
} T_DEVREQ;
```

I indicates an input parameter and O an output parameter. Input parameters must not be changed by the device driver. Parameters other than input parameters (I) are initially cleared to 0 by device management. After that, device management does not modify them. next is used to link the request packet. In addition to being used for keeping track of request packets in device management, it is also used by the completion wait function (waitfn) and abort function (abortfn).

exinf can be used freely by the device driver to store any other information. Device management does not pay attention to its contents.

The device ID of the device to which the request is issued is specified in devid.

The request command is specified in cmd as follows.

cmd := (TDC_READ || TDC_WRITE)

```
#define  TDC_READ    1  /* read request */
#define  TDC_WRITE   2  /* write request */
```

If abort processing is to be carried out, abort is set to TRUE right before calling the abort function (abortfn). abort is a flag indicating whether abort processing was requested and does not indicate that processing was aborted. In some cases, abort is set to TRUE even when the abort function (abortfn) is not called. Abort processing is performed when a request with abort set to TRUE is actually passed to the device driver.

The start and size parameters for the tk_rea_dev and tk_wri_dev calls are set to the values specified here.

Similarly, the buf parameter for tk_rea_dev or tk_wri_dev calls are set to the value specified here.

In asize, the device driver sets the value returned in asize by tk_wai_dev.

In error, the device driver sets the error code returned by tk_wai_dev in its return code. E_OK indicates a normal result.

[Difference with T-Kernel]

Specifications concerning virtual memory and user space do not exist, so nolock and tskspc do not exist among T_DEVREQ members. Also, the reserved member rsv does not exist so that the different bit widths of the INT type on different targets can be absorbed.

[Difference with T-Kernel 1.00.00]

The type of start, size, and asize is W instead of INT.

[Porting Guideline]

The specifications concerning virtual memory do not exist in µT-Kernel, so all memory is treated as actual memory. Thus, when a device driver created for µT-Kernel is used in the systems with T-Kernel's virtual memory, the application should make the memory resident.

Similarly, the specification concerning user space does not exist in µT-Kernel. So when device drivers created for µT-Kernel are ported to the T-Kernel, they should be implemented so that the buffer area prepared by the application can be allocated in shared space.

- Open Function: ER openfn(ID devid, UINT omode, VP exinf)

devid	Device ID of the device to open
omode	Open mode (same as tk_opn_dev)
exinf	Extended information set at device registration
return code	Error

  The open function openfn is called when tk_opn_dev is invoked.

  The function openfn performs processing to enable the use of a device. Details of the processing are device-dependent; if no processing is needed, it does nothing. The device driver does not need to remember whether a device is open or not, nor is it necessary to treat as error the calling of another processing function only because the device was not opened (openfn had not been called). If another processing function is called for a device that is not open, the necessary processing may be performed as long as there is no problem in device driver operation.

  When openfn is used to perform operations such as device initialization, in principle, no processing should be performed that causes a WAIT state. The processing and return from openfn must be as prompt as possible. In the case of a device such as a serial port for which it is necessary to set the communication mode, for example, the device can be initialized when the communication mode is set by tk_wri_dev. In such cases, there is no need for openfn to initialize the device.

  When the same device is opened multiple times, normally this function is called only the first time. If, however, the driver attribute TDA_OPENREQ is specified in device registration, this function is called each time the device is opened.

  Since processing related to the mode of opening and the opening of the device multiple times is handled by device management, processing related to them is not required in the openfn function. Similarly, omode is simply passed as reference information; no processing related to omode is required.

- Close Function: ER closefn(ID devid, UINT option, VP exinf)

devid	Device ID of the device to close

option	Close option (same as tk_cls_dev)
exinf	Extended information set at device registration
return code	Error

The close function closefn is called when tk_cls_dev is invoked.

The closefn function performs processing to end use of a device. Details of the processing are device-dependent; if no processing is needed, it does nothing.

If the device is capable of ejecting media and TD_EJECT is set in option, media ejection is performed.

When closefn is used to perform device shutdown processing or media ejection, in principle, no processing should be performed that causes a WAIT state. The processing and return from closefn must be as prompt as possible. If media ejection takes time, control may be returned from closefn without waiting for the ejection to complete.

When the same device is opened multiple times, normally this function is called only the last time it is closed. If, however, the driver attribute TDA_OPENREQ is specified in device registration, this function is called each time the device is closed. In this case TD_EJECT is specified in option only the last time.

Since processing related to the mode of opening and the opening of the device multiple times is handled by device management, processing related to them is not required in the closefn function.

- Processing Start Function: ER execfn(T_DEVREQ *devreq, TMO tmout, VP exinf)

devreq	Request packet
tmout	Request acceptance timeout (ms)
exinf	Extended information set at device registration
return code	Error

The execfn function is called when tk_rea_dev or tk_wri_dev is invoked and starts the processing requested in devreq. This function only starts the requested processing, returning to its caller without waiting for the processing to complete. The time required to start processing depends on the device driver; this function does not necessarily complete immediately.

When new processing cannot be accepted, this function goes to WAIT state for request acceptance. If the new request cannot be accepted within the time specified in tmout, the function times out.

The attribute TMO_POL or TMO_FEVR can be specified in tmout. If the function times out, E_TMOUT is passed in the execfn return code. The timeout applies only to request acceptance and not to the processing after acceptance.

When error is passed in the execfn return code, the request is considered not to have been accepted and the request packet is discarded.

If processing is aborted before the request is accepted (before the requested processing starts), E_ABORT is passed in the execfn return code and the request packet is discarded. If processing is aborted after the processing has been accepted, E_OK is returned for this function. The request packet is not discarded until waitfn is executed and processing completes.

When processing is aborted, the important thing is to return from execfn as quickly as possible. If processing will end soon anyway without aborting, it is not necessary to abort.

- Completion Wait Function: INT waitfn(T_DEVREQ *devreq, INT nreq, TMO tmout, VP exinf)

devreq	Request packet list
nreq	Request packet count
tmout	Timeout (ms)
exinf	Extended information set at device registration
return code	Completed request packet number or error

The waitfn function is called when tk_wai_dev is invoked.

devreq is a list of request packets in a chain linked by devreq->next. This function waits for completion of any of the nreq request packets starting from devreq. The final next is not necessarily NULL, so the count passed in nreq must always be followed. The number of the completed request packets (which one after devreq) is passed in the return code. The first one is numbered 0 and the last one is numbered nreq-1. Here completion means any of normal completion, abnormal (error) termination, or abort.

The time to wait until completion is set in tmout. TMO_POL or TMO_FEVR can be specified as the tmout attribute. If the wait times out, the requested processing continues. The waitfn return code in case of timeout is E_TMOUT. The error parameter of the request packet does not change. Note that if return from waitfn occurs while the requested processing continues, error must be returned in the waitfn return code but the processing must be completed even when error is passed in the return code, and a value other than error must not be returned if processing is ongoing. As long as error is not passed in the waitfn return code, the request is considered to be pending and no request packet is discarded. When the number of a request packet whose processing was completed is passed in the waitfn return code, the processing of that request is considered to be completed and that request packet is discarded.

IO error and other device-related errors are stored in the error parameter of the request packet. Error is passed in the waitfn return code when completion waiting did not take place properly. The waitfn return code is set in the tk_wai_dev return code, whereas the request packet error value is returned in ioer.

Abort processing differs depending on whether the wait is for completion of a single request (nreq = 1) or multiple requests (nreq > 1). When completion of a single request is being waited for, the request currently being processed is aborted. When waiting for completion of multiple requests, only the wait is aborted (wait release), not the requested processing itself. When a wait for multiple requests is aborted (wait release), E_ABORT is passed in the waitfn return code.

During a wait for request completion, an abort request may be set in the abort parameter of a request packet. In such a case, if it is a single request, the request abort processing must be performed. If the wait is for multiple requests, it is also preferable that abort processing be executed, but it is also possible to ignore the abort flag.

When abort occurs, the important thing is to return from waitfn as quickly as possible. If processing will end soon anyway without aborting, it is not necessary to abort.

As a rule, E_ABORT is returned in the request packet error parameter when processing is aborted; but a different error code may be returned as appropriate based on the device properties. Returning E_OK on the basis that the processing right up to the abort is valid is also allowed. If processing completes normally to the end, E_OK is returned even if there was an abort request.

- Abort Function: ER abortfn(ID tskid, T_DEVREQ *devreq, INT nreq, VP exinf)

`tskid`	Task ID of the task executing `execfn` or `waitfn`
`devreq`	Request packet list
`nreq`	Request packet count
`exinf`	Extended information set at device registration
return code	Error

The function `abortfn` causes `execfn` or `waitfn` to return promptly when the specified request is being executed. Normally this means the request being processed is aborted. If, however, the processing can be completed soon without aborting, it may not have to be aborted. The important thing is to return as quickly as possible from `execfn` or `waitfn`.

`tskid` indicates the task executing the request specified in `devreq`. In other words, it is the task executing `execfn` or `waitfn`. `devreq` and `nreq` are the same as the parameters that were passed to `execfn` or `waitfn`. In the case of `execfn`, `nreq` is always 1.

`abortfn` is called by a different task from the one executing `execfn` or `waitfn`. Since both tasks run concurrently, mutual exclusion control must be performed as necessary. It is possible that the `abortfn` function will be called immediately before calling `execfn` or `waitfn`, or during return from these functions. Measures must be taken to ensure proper operation in such cases. Before `abortfn` is called, the `abort` flag in the request packet whose processing is to be aborted is set to TRUE, enabling `execfn` or `waitfn` to know whether there is going to be an abort request.

When `waitfn` is executing for multiple requests (`nreq` > 1), this is treated as a special case differing as follows from other cases.

- Only the completion wait is aborted (wait release), not the requested processing.
- The `abort` flag is not set in the request packet (remains as `abort` = FALSE).

Aborting a request when `execfn` and `waitfn` are not executing is done not by calling `abortfn` but by setting the `abort` flag in the request packet. If `execfn` is called when the `abort` flag is set, the request is not accepted. If `waitfn` is called, abort processing is the same as that when `abortfn` is called.

If a request for which processing was started by `execfn` is aborted before `waitfn` was called to wait for its completion, the completion of the aborted processing is notified when `waitfn` is called. Even though processing was aborted, the request itself is not discarded until its completion has been confirmed by `waitfn`.

`abortfn` only starts abort processing returning promptly without waiting for the abort to complete.

`abortfn` is called in the following cases.

- When a device is being closed by `tk_cls_dev` or by subsystem cleanup processing, and a device descriptor was processing a request, `abortfn` is used to abort the request being processed by that device descriptor.

- Event Handling Function: INT `eventfn`(INT `evttyp`, VP evtinf, VP `exinf`)

`evttyp`	Driver request event type
`evtinf`	Information for each event type
`exinf`	Extended information set at device registration
return code	Return code defined for each event type or error

The following driver request event types are defined. Those with positive values are called by `tk_evt_dev,` and those with negative values are called inside device management.

```
#define  TDV_SUSPEND  (-1) /* suspend                              */
#define  TDV_RESUME   (-2) /* resume                               */
#define  TDV_CARDEVT  1    /* PC Card event (see Card Manager      */
#define  TDV_USBEVT   2    /* USB event (see USB Manager)          */
```

The processing performed by an event function is defined for each event type. Suspend and resume processing are discussed later below.

When a device event is called by `tk_evt_dev`, the `eventfn` return code is set as the `tk_evt_dev` return code.

Requests to event functions must be accepted even if another request is processing and must be processed as quickly as possible.

4.10.5 Attribute Data

Attribute data is classified broadly into the following three kinds of data.

- **Common attributes**

  Attributes defined in common for all devices (device drivers).

- **Device kind attributes**

  Attributes defined in common for devices (device drivers) of the same kind.

- **Device-specific attributes**

  Attributes defined independently for each device (device driver).

For the device kind attributes and device-specific attributes, see the specifications for each device. Only the common attributes are defined here.

Common attributes are assigned attribute data numbers in the range from -1 to -99. While common attribute data numbers are the same for all devices, not all devices necessarily support all common attributes. If an unsupported data number is specified, error code `E_PAR` is returned.

```
#define  TDN_EVENT     (-1) /* RW: event notification message buffer ID */
#define  TDN_DISKINFO  (-2) /* R: disk information                      */
#define  TDN_DISPSPEC  (-3) /* R: display device specification          */
RW: read (tk_rea_dev)/write (tk_wri_dev) enabled
R: read (tk_rea_dev) only
```

`TDN_EVENT`: Event Notification Message Buffer ID

Data type `ID`

The ID of the message buffer used for device event notification. Since the system default message buffer ID is passed in device registration, that ID is set as the initial setting when a driver is started.

If 0 is set, device events are not notified.

Device event notification is discussed later below.

`TDN_DISKINFO`: Disk Information

Data type: `DiskInfo`

```
typedef  enum {
         DiskFmt_STD    = 0, /* standard (HD, etc.)    */
         DiskFmt_2DD    = 1, /* 2DD 720KB              */
         DiskFmt_2HD    = 2, /* 2HD 1.44MB             */
         DiskFmt_CDROM  = 4, /* CD-ROM 640MB           */
} DiskFormat;

typedef  struct {
         DiskFormat  format;       /* format                 */
         UW          protect:1;    /* protected status       */
         UW          removable:1;  /* removable              */
         UW          rsv:30;       /* reserved (always 0)    */
         W           blocksize;    /* block size in bytes    */
         W           blockcount;   /* total block count      */
} DiskInfo;
```

TDN_DISPSPEC: Display Device Specification

Data type: DEV_SPEC

```
typedef struct {
H  attr;          /* device attributes             */
H  planes;        /* number of planes              */
H  pixbits;       /* pixel bits (boundary/valid)   */
H  hpixels;       /* horizontal pixels             */
H  vpixels;       /* vertical pixels               */
H  hres;          /* horizontal resolution         */
H  vres;          /* vertical resolution           */
H  color[4];      /* color information             */
H  resv[6];       /* reserved                      */
} DEV_SPEC;
```

4.10.6 Device Event Notification

A device driver sends events occurring in devices to the event notification message buffer (TDN_EVENT) as device event notification. When registering devices, whether the system specifies the default message buffer for event notification or not is implementation-defined.

The following event types are defined.

```
typedef  enum tdevttyp {
         TDE_unknown      = 0,     /* undefined                  */
         TDE_MOUNT        = 0x01,  /* media mounted              */
         TDE_EJECT        = 0x02,  /* media ejected              */
         TDE_ILLMOUNT     = 0x03,  /* media illegally mounted    */
         TDE_ILLEJECT     = 0x04,  /* media illegally ejected    */
         TDE_REMOUNT      = 0x05,  /* media remounted            */
         TDE_CARDBATLOW   = 0x06,  /* card battery alarm         */
         TDE_CARDBATFAIL  = 0x07,  /* card battery failure       */
         TDE_REQEJECT     = 0x08,  /* media eject request        */
```

```
            TDE_PDBUT        = 0x11,  /* PD button state change      */
            TDE_PDMOVE       = 0x12,  /* PD move                     */
            TDE_PDSTATE      = 0x13,  /* PD state change             */
            TDE_PDEXT        = 0x14,  /* PD extended event           */
            TDE_KEYDOWN      = 0x21,  /* key down                    */
            TDE_KEYUP        = 0x22,  /* key up                      */
            TDE_KEYMETA      = 0x23,  /* meta key state change       */
            TDE_POWEROFF     = 0x31,  /* power switch off            */
            TDE_POWERLOW     = 0x32,  /* low power alarm             */
            TDE_POWERFAIL    = 0x33,  /* power failure               */
            TDE_POWERSUS     = 0x34,  /* auto suspend                */
            TDE_POWERUPTM    = 0x35,  /* clock update                */
            TDE_CKPWON       = 0x41   /* auto power on notification  */
     } TDEvttyp;
```

Device events are notified in the following format. The contents of event notification and size differ with each event type.

```
     typedef struct t_devevt {
           TDEvttyp evttyp; /* event type */
           /* Information specific to each event type is appended here. */
     } T_DEVEVT;
```

The format of device event notification with device ID is as follows.

```
     typedef struct t_devevt_id {
           TDEvttyp evttyp; /* event type */
           ID devid; /* device ID */
           /* Information specific to each event type is appended here. */
     } T_DEVEVT_ID;
```

See the device driver specifications for event details.

Measures must be taken so that if event notification cannot be sent because the message buffer is full, the lack of notification will not adversely affect operation on the receiving end. One option is to hold the notification until space becomes available in the message buffer, but in that case other device driver processing should not, as a rule, be allowed to fall behind as a result. Processing on the receiving end should be designed to the extent possible to avoid message buffer overflow.

4.10.7 Device Suspend/Resume Processing

Device drivers suspend and resume device operations in response to the issuing of suspend/ resume (`TDV_USPEND`/ `TDV_RESUME`) events to the event handling function (`eventfn`). Suspend and resume events are issued only to physical devices.

`TDV_USPEND`: Suspend Device

`evttyp` = TDV_USPEND

`evtinf` = NULL (none)

Suspension processing takes place in the following steps.

(A) If there is a request being processed at the time, the device driver waits for it to complete, pauses it or aborts. Which of these options to take depends on the device driver implementation. However, since the suspension must be effected as quickly as possible, pause or abort should be chosen if completion of the request will take time. Suspend events can be issued only for physical devices, but the same processing is applied to all logical devices included in the physical device.

Pause: Processing is suspended, then continues after the device resumes operation.

Abort: Processing is aborted just as when the abort function (`abortfn`) is executed, and is not continued after the device resumes operation.

(B) New requests other than a resume event are not accepted.

(C) The device power is cut and other suspension processing is performed.

Abort should be avoided, if at all possible, because of its effects on applications. It should be used only in such cases as long input waits from a serial port, or when interruption would be difficult. Normally, it is best to wait for completion of a request or, if possible, to pause (suspension and resumption).

Requests arriving at the device driver in suspend state are made to wait until operation resumes after which acceptance processing is performed. If the request does not involve access to the device, however, or can otherwise be processed even during suspension, a request may be accepted without waiting for resumption.

`TDV_RESUME`: Resume Device

```
evttyp = TDV_RESUME
evtinf = NULL (none)
```

Resumption processing takes place as follows.

(A) The device power is turned back on, the device states are restored and other device resumption processing is performed.

(B) Paused processing is resumed.

(C) Request acceptance is resumed.

4.10.8 Special Properties of Disk Devices

Since there is no virtual memory function in µT-Kernel, the disk driver does not require special handling.

[Difference with T-Kernel]

For systems with virtual memory like T-Kernel, the disk device is special. When utilizing virtual memory, data transfer is executed between memory and disk. So the disk driver needs to be called by the OS.

The need for the OS to perform data transfer with a disk arises when access is made to nonresident memory and the memory contents must be read from the disk (page in). The OS calls the disk driver in this case.

If nonresident memory is accessed in the disk driver, the OS must likewise call the disk driver. In such a case, if the disk driver treats the access to nonresident memory as a wait for page in, it is possible that the OS will again request disk access. Even then, the disk driver must be able to execute the later OS request.

A similar case may arise in suspension processing. When access is made to nonresident memory during suspension processing and a disk driver is called, if that disk driver is already suspended, page in will not be possible. To avoid such a situation, suspension processing should suspend other devices before disk devices. If there are multiple disk devices, however, the order of their suspension is indeterminate. For this reason, during suspension processing, a disk driver must not access nonresident memory.

Because of the above limitations, a disk driver must not use (access) nonresident memory. It is possible, however, that the IO buffer (`buf`) space specified with `tk_rea_dev` or `tk_wri_dev` will be nonresident memory, since this is a memory location specified by the caller. In the case of IO buffers, therefore, it is necessary to make the memory space resident at the time of IO access.

4.11 Subsystem and Device Driver Starting

In µT-Kernel, there is no particular definition concerning the start-up of the subsystem and device driver. In terms of portability, it is desirable to prepare the entry routine in the same format as T-Kernel if possible.

[Difference with T-Kernel]

In T-Kernel, subsystems and device drivers have the following entries:

Entry routines like the following are defined for subsystems and device drivers.

```
ER main( INT ac, UB *av[] )
{
      if ( ac >= 0 ) {
      /* Subsystem/device driver start processing */
      } else {
      /* Subsystem/device driver termination processing */
      }
      return ercd;
}
```

This entry routine simply performs startup processing or termination processing for a subsystem or device driver and does not provide any actual service. It must return to its caller as soon as the startup processing or termination processing is performed. An entry routine must perform its processing as quickly as possible and return to its caller.

An entry routine is called by the task which belongs to the system resource group at the time of normal system startup or shutdown, and runs in the context of the OS start processing task or termination processing task (protection level 0). In a system that supports dynamic loading of subsystems and device drivers, it may be called at other times besides system startup and shutdown.

When there are multiple subsystems and device drivers, each of the entry routines is called one at a time at system startup and shutdown. In no case are multiple entry routines called by different tasks at the same time. Accordingly, if subsystem or device driver initialization needs to be performed in a certain order, this order can be maintained by completing all necessary processing before returning from an entry routine.

The entry routine function name is normally `main`, but any other name may be used if, for example, main cannot be used because of linking with the OS.

- Startup processing

`ac`	Number of parameters (>= 0)
`av`	Parameters (string)
return code	Error

  A value of `ac` >= 0 indicates startup processing. After performing the subsystem or device driver initialization, it registers the subsystem or device driver.

  If a negative value (error) is returned as a return value, the boot process is assumed to have failed. In this case, subsystems and device drivers are deleted from the memory. So an error should not be returned while registering subsystems and device drivers. The registration must first be removed before returning the error. Allocated resources must also be released. They are not released automatically.

  The parameters `ac` and `av` are the same as the parameters passed to the standard C language `main`()

function, with ac indicating the number of parameters and av indicating a parameter string as an array of ac + 1 pointers. The array termination (av[ac]) is NULL.

av[0] is the name of the subsystem or device driver. Generally this is the file name of the subsystem or device driver, but the kind of name in which it is stored is implementation-dependent. It is also possible to have no name (blank string "").

Parameters from av[1] onwards are defined separately for each subsystem and device driver.

After exiting from the entry routine, the character string space specified by av is deleted, so parameters must be saved to a different location if needed.

- Termination processing

ac	-1
av	NULL
return code	Error

A value of ac < 0 indicates termination processing. After deleting the subsystem or device driver registration, the entry routine releases allocated resources. If an error occurs during termination processing, the processing must not be aborted but must be completed to the extent possible. If some of the processing could not be completed normally, an error is passed in the return code.

The behavior if termination processing is called while requests to the subsystem or device driver are being processed is dependent on the subsystem or device driver implementation. Generally, termination processing is called at system shutdown and requests are not issued during processing. For this reason, ordinary behavior is not guaranteed in the case of requests issued during termination processing.

4.12 Debugger Support Functions

Debugger support functions provide functions allowing a debugger to reference µT-Kernel internal states and run a trace. The functions provided by debugger support functions are only for debugger use and not for use by applications or other programs.

The debugger support function is a function for debugging, so whether it is implemented or not depends on the OS provider's judgment. However, when implementing it, the interface specified in the specification should be observed and the same function as that specified in the specification should be implemented. Also, if other functions are provided for debugging, their names should be different from those specified in the specification.

[General cautions and notes]

- Except where otherwise noted, debugger support functions system calls (`td_xxx`) can be called from a task independent portion and while dispatching and interrupts are disabled. There may be some limitations, however, imposed by specific implementations.
- When debugger support functions system calls (`td_xxx`) are invoked in interrupts disabled state, they are processed without enabling interrupts. Other OS states likewise remain unchanged during this processing. Changes in OS states may occur if a system call is invoked while interrupts or dispatching are enabled, since the OS continues operating.
- Error codes such as `E_PAR` and `E_CTX` that always have the possibility of occurring are not described here unless there is some special reason for doing so.
- Detection of `E_PAR` and `E_CTX` is implementation-defined. The call may not be detected as an error, so you should not invoke those calls which can cause such an error.

4.12.1 Kernel Internal State Reference Functions

These functions enable a debugger to get µT-Kernel internal states. They include functions for getting a list of objects, getting task precedence, getting the order in which tasks are queued, getting the status of objects, system and task registers, and getting time.

td_lst_tsk, td_lst_sem, td_lst_flg, td_lst_mbx td_lst_mtx, td_lst_mbf, td_lst_por, td_lst_mpf td_lst_mpl, td_lst_cyc, td_lst_alm, td_lst_ssy

Reference Object ID List

[C Language Interface]

```
INT ct = td_lst_tsk  ( ID list[], INT nent ) ;/* task */
INT ct = td_lst_sem  ( ID list[], INT nent ) ; /* semaphore */
INT ct = td_lst_flg  ( ID list[], INT nent ) ; /* event flag */
INT ct = td_lst_mbx  ( ID list[], INT nent ) ; /* mailbox */
INT ct =  td_lst_mtx ( ID list[], INT nent ) ; /* mutex */
INT ct =  td_lst_mbf ( ID list[], INT nent ) ; /* message buffer */
INT ct =  td_lst_por ( ID list[], INT nent ) ; /* rendezvous port */
INT ct = td_lst_mpf  ( ID list[], INT nent ) ; /* fixed-size memory pool */
INT ct =  td_lst_mpl ( ID list[], INT nent ) ; /* variable-size memory pool */
INT ct = td_lst_cyc  ( ID list[], INT nent ) ; /* cyclic handler */
INT ct = td_lst_alm  ( ID list[], INT nent ) ; /* alarm handler */
INT ct = td_lst_ssy  ( ID list[], INT nent ) ; /* subsystem */
```

[Parameters]

ID	list[]	Location of object ID list
INT	nent	Maximum number of list entries to retrieve

[Return Parameters]

INT	ct	Number of objects used
	or	Error Code

[Description]

Gets a list of IDs of objects currently being used, and puts up to `nent` IDs into `list`. The number of objects used is passed in the return code. If return code > `nent`, this means that not all IDs could be retrieved in this system call.

td_rdy_que

Get Task Precedence

[C Language Interface]

```
INT ct = td_rdy_que ( PRI pri, ID list[], INT nent ) ;
```

[Parameters]

PRI	pri	Task priority
ID	list[]	Location of task ID list
INT	nent	Maximum number of list entries

[Return Parameters]

INT	ct	Number of priority pri tasks in a run state
	or	Error Code

[Description]

Gets a list of IDs of the tasks in a run state (READY state or RUN state) whose task priority is `pri`, arranged in order from highest to lowest precedence.

This function stores in the location designated in `list` up to `nent` task IDs, arranged in order of precedence starting from the highest-precedence task ID at the head of the list.

The number of tasks in a run state with priority `pri` is passed in the return code. If return code > `nent`, this means that not all task IDs could be retrieved in this call.

td_sem_que, td_flg_que, td_mbx_que, td_mtx_que td_smbf_que, td_rmbf_que, td_cal_que, td_acp_que td_mpf_que, td_mpl_que

Reference Queue

[C Language Interface]

```
INT ct = td_sem_que  ( ID semid, ID list[], INT nent ) ;   /* semaphore              */
INT ct = td_flg_que  ( ID flgid, ID list[], INT nent ) ;   /* event flag             */
INT ct = td_mbx_que  ( ID mbxid, ID list[], INT nent ) ;   /* mailbox                */
INT ct = td_mtx_que  ( ID mtxid, ID list[], INT nent ) ;   /* mutex                  */
INT ct = td_smbf_que ( ID mbfid, ID list[], INT nent ) ;   /* message buffer send    */
INT ct = td_rmbf_que ( ID mbfid, ID list[], INT nent ) ;   /* message buffer receive */
INT ct = td_cal_que  ( ID porid, ID list[], INT nent ) ;   /* rendezvous call        */
INT ct = td_acp_que  ( ID porid, ID list[], INT nent ) ;   /* rendezvous accept      */
INT ct = td_mpf_que  ( ID mpfid, ID list[], INT nent ) ;   /* fixed-size memory pool */
INT ct = td_mpl_que  ( ID mplid, ID list[], INT nent ) ;   /* variable-size memory pool*/
```

[Parameters]

ID	--id	Object ID
ID	list[]	Location of waiting task IDs
INT	nent	Maximum number of list entries

[Return Parameters]

INT	ct	Number of waiting tasks
	or	Error Code

[Error Codes]

E_ID	Bad identifier
E_NOEXS	Object does not exist

[Description]

Gets a list of IDs of tasks waiting for the object designated in --id.

This function stores in the location designated in `list` up to `nent` task IDs, arranged in the order in which tasks are queued, starting from the first task in the queue. The number of queued tasks is passed in the return code. If return code > `nent`, this means that not all task IDs could be retrieved in this call.

td_ref_tsk

Reference Task State

[C Language Interface]

```
ER ercd = td_ref_tsk ( ID tskid, TD_RTSK *rtsk );
```

[Parameters]

ID	tskid	Task ID (TSK_SELF can be specified)
TD_RTSK	rtsk	Address of Packet for returning the task state

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Bad identifier
E_NOEXS	Object does not exist

[Description]

Gets the state of the task designated in tskid. This function is similar to tk_ref_tsk, with the task start address and stack information added to the state information obtained.

```
typedef struct  td_rtsk {
     VP      exinf;      /* extended information             */
     PRI     tskpri;     /* current priority                 */
     PRI     tskbpri;    /* base priority                    */
     UINT    tskstat;    /* task state                       */
     UW      tskwait;    /* wait factor                      */
     ID      wid;        /* waiting object ID                */
     INT     wupcnt;     /* queued wakeup request count      */
     INT     suscnt;     /* SUSPEND request nesting count    */
     FP      task;       /* task start address               */
     W       stksz;      /* user stack size (in bytes)       */
     VP      istack;     /* user stack pointer initial value */
} TD_RTSK;
```

The stack area extends from the stack pointer initial value toward the low addresses for the number of bytes designated as the stack size.

istack - stksz ≤ user stack area < istack

Note that the stack pointer initial value (istack) 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 current value of the stack pointer.

[Difference with T-Kernel]

Maximum continuous execution time (slicetime), disabled waiting factor (waitmask), permitted task exception (texmask), and occurred task event (tskevent) do not exist in td_rtsk. This is because µT-Kernel does not have functions related to timesharing execution, wait disabled, task exception, and task event.

For the same reason, TTS_NODISWAI (non-disabled wait state) does not exist in the values set to tskstat, and the value relating to task event (TTW_EVn) does not exist in the values set to tskwait.

The members (sstksz, isstack) relating to the system stack do not exist. This is because in µT-Kernel, it is assumed that each task has one stack.

[Difference with T-Kernel 1.00.00]

stksz, the member of TD_RTSK, is of W type instead of INT type.

tskwait, the member of TD_RTSK, is of UW type instead of UINT type. In µT-Kernel, 16-bit width is sufficient for this member, but 32-bit width is required in the T-Kernel. Therefore, this member is modified to UW type based on the policy that those fields that require 32-bit width are set to the W/UW types since their bit widths are specifically defined.

td_ref_sem, td_ref_flg, td_ref_mbx, td_ref_mtx td_ref_mbf, td_ref_por, td_ref_mpf, td_ref_mpl td_ref_cyc, td_ref_alm, td_ref_ssy

Reference Queue

[C Language Interface]

```
ER ercd = td_ref_sem  ( ID semid, TD_RSEM *rsem );  /* semaphore              */
ER ercd = td_ref_flg  ( ID flgid, TD_RFLG *rflg );  /* event flag             */
ER ercd = td_ref_mbx  ( ID mbxid, TD_RMBX *rmbx );  /* mailbox                */
ER ercd = td_ref_mtx  ( ID mtxid, TD_RMTX *rmtx );  /* mutex                  */
ER ercd = td_ref_mbf  ( ID mbfid, TD_RMBF *rmbf );  /* message buffer         */
ER ercd = td_ref_por  ( ID porid, TD_RPOR *rpor );  /* rendezvous port        */
ER ercd = td_ref_mpf  ( ID mpfid, TD_RMPF *rmpf );  /* fixed-size memory      */
ER ercd = td_ref_mpl  ( ID mplid, TD_RMPL *rmpl );  /* variable-size memo     */
ER ercd = td_ref_cyc  ( ID cycid, TD_RCYC *rcyc );  /* cyclic handler         */
ER ercd = td_ref_alm  ( ID almid, TD_RALM *ralm );  /* alarm handler          */
ER ercd = td_ref_ssy  ( ID ssid, TD_RSSY *rssy );   /* subsystem pool ry pool */
```

[Parameters]

```
ID        --id      Object ID
TD_R--    r--       Address of status information packet
```

[Return Parameters]

```
ER        ercd      Error code
```

[Error Codes]

`E_OK`	Normal completion
`E_ID`	Bad identifier
`E_NOEXS`	Object does not exist

[Description]

Gets the status of an object. This is similar to tk_ref -. The return packets are defined as follows.

```
/*
*Semaphore status information   td_ref_sem
*/
typedef struct td_rsem {
        VP      exinf;   /* extended information         */
        ID      wtsk;    /* waiting task ID              */
        INT     semcnt;  /* current semaphore count      */
} TD_RSEM;
/*
```

```
*Event flag status information  td_ref_flg
*/
typedef struct td_rflg {
        VP      exinf;   /* extended information         */
        ID      wtsk;    /* waiting task ID              */
        UINT    flgptn;  /* current event flag pattern   */
} TD_RFLG;

/*
*Mailbox        status information   td_ref_mbx
*/
typedef struct td_rmbx {
        VP      exinf;   /* extended information         */
        ID      wtsk;    /* waiting task ID              */
        T_MSG   *pk_msg; /* next message to be received  */
} TD_RMBX;

/*
*Mutex status information td_ref_mtx
*/
typedef struct td_rmtx {
        VP      exinf;   /* extended information         */
        ID      htsk;    /* locking task ID              */
        ID      wtsk;    /* ID of task waiting for lock */
} TD_RMTX;

/*
* Message buffer status information   td_ref_mbf
*/
typedef struct td_rmbf {
        VP      exinf;   /* extended information                        */
        ID      wtsk;    /* receive waiting task ID                     */
        ID      stsk;    /* send waiting task ID                        */
        INT     msgsz;   /* size (in bytes) of next message to be received  */
        W       frbufsz; /* free buffer size (in bytes)                 */
        INT     maxmsz;  /* maximum message length (in bytes)           */
} TD_RMBF;

/*
*Rendezvous port status  information   td_ref_por
*/
typedef struct td_rpor {
        VP      exinf;   /* extended information                       */
        ID      wtsk;    /* call waiting task ID                       */
        ID      atsk;    /* acceptance waiting task ID                 */
        INT     maxcmsz; /* call message maximum length (in bytes)    */
        INT     maxrmsz; /* accept message maximum length (in bytes)  */
} TD_RPOR;

/*
*Fixed-size memory pool status information   td_ref_mpf
*/
typedef struct td_rmpf {
        VP      exinf;   /* extended information         */
        ID      wtsk;    /* waiting task ID              */
        W       frbcnt;  /* free block count             */
} TD_RMPF;

/*
*Variable-size memory pool status information  td_ref_mpl
*/
typedef struct td_rmpl {
        VP      exinf;   /* extended information                      */
        ID      wtsk;    /* waiting task ID                           */
        W       frsz;    /* total free space (in bytes)               */
        W       maxsz;   /* maximum contiguous free space (in bytes)  */
} TD_RMPL;

/*
*Cyclic handler status information  td_ref_cyc
*/
typedef struct td_rcyc {
        VP      exinf;   /* extended information                      */
        RELTIM  lfttim;  /* time remaining until next handler start   */
        UINT    cycstat; /* cyclic handler status                     */
} TD_RCYC;
]
/*
*Alarm handler status information   td_ref_alm
*/
typedef struct td_ralm {
        VP      exinf;   /* extended information                      */
        RELTIM  lfttim;  /* time remaining until next handler start   */
        UINT    almstat; /* alarm handler status                      */
} TD_RALM;

/*
*Subsystem status information  td_ref_ssy
*/
typedef struct td_rssy {
        PRI     ssypri;  /* subsystem priority                        */
        W       resblksz; /* resource control block size (in bytes)   */
} TD_RSSY;
```

[Difference with T-Kernel 1.00.00]

`frbufsz`, the member of `TD_RMBF`, is of W type instead of INT type.

`frbcnt`, the member of `TD_RMPF`, is of W type instead of INT type.

`frsz` and `maxsz`, the members of `TD_RMPL`, are of W type instead of INT type.

`resblksz`, the member of `TD_RSSY`, is of W type instead of INT type.

td_inf_tsk

Reference Task Statistics

[C Language Interface]

```
ER ercd = td_inf_tsk ( ID tskid, TD_ITSK *pk_itsk, BOOL clr );
```

[Parameters]

ID	tskid	Task ID (TSK_SELF can be designated)
TD_ITSK*	pk_itsk	Address of packet for returning task statistics
BOOL	clr	Task statistics clear flag

[Return Parameters]

ER ercd Error code

pk_itsk detail:

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

[Error Codes]

E_OK	Normal completion
E_ID	ID number is invalid
E_NOEXS	Object does not exist

[Description]

Gets task statistics. When clr = TRUE (= 0), accumulated information is reset (cleared to 0) after getting the statistics.

td_get_reg

Get Task Register

[C Language Interface]

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

[Parameters]

```
ID          tskid Task ID (TSK_SELF cannot be designated)
```

[Return Parameters]

T_REGS*	pk_regs	General registers
T_EIT*	pk_eit	Registers saved when exception is raised
T_CREGS*	pk_cregs	Control registers
ER	ercd	Error code

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

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task designated in tskid does not exist)
E_OBJ	Invalid object state (issued for current RUN state task)

[Description]

Gets the register values of the task designated in `tskid`. This is similar to `tk_get_reg`.

Registers cannot be referenced for the task currently in RUN state. Except when a task-independent portion is executing, the current RUN state task is the invoking task.

When NULL is designated for pk_`regs`, pk_`eit`, or pk_`cregs`, the corresponding register is not referenced.

The contents of T_REGS, T_EIT, and T_CREGS are implementation-dependent.

td_set_reg

Set Task Register

[C Language Interface]

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

[Parameters]

ID	tskid	Task ID (TSK_SELF cannot be specified)
T_REGS*	pk_regs	General registers
T_EIT*	pk_eit	Registers saved when exception is raised
T_CREGS*	pk_cregs	Control registers

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

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_ID	Invalid ID number (tskid is invalid or cannot be used)
E_NOEXS	Object does not exist (the task designated in tskid does not exist)
E_OBJ	Invalid object state (issued for current RUN state task)

[Description]

Sets registers of the task designated in tskid. This is similar to tk_set_reg.

Registers cannot be set for the task currently in RUN state. Except when a task-independent portion is executing, the current RUN state task is the invoking task.

When NULL is designated for pk_regs, pk_eit, or pk_cregs, the corresponding register is not set.

The contents of T_REGS, T_EIT, and T_CREGS are implementation-dependent.

td_ref_sys

Reference System Status

[C Language Interface]

```
ER ercd = td_ref_sys ( TD_RSYS *pk_rsys ) ;
```

[Parameters]

```
TD_RSYS*    pk_rsys     Address of packet for returning status information
```

[Return Parameters]

```
ER          ercd        Error code
```

```
pk_rsys detail:
    INT     sysstat     System status
    ID      runtskid    ID of current RUN state task
    ID      schedtskid  ID of task scheduled to go to RUN state
```

[Error Codes]

E_OK	Normal completion

[Description]

Gets the system status. This is similar to `tk_ref_sys`.

td_get_tim

Get System Time

[C Language Interface]

```
ER ercd = td_get_tim ( SYSTIM *tim, UW *ofs ) ;
```

[Parameters]

SYSTIM*	tim	Address of packet for returning current time (ms)
UW*	ofs	Location for returning elapsed time from tim (nanoseconds)

[Return Parameters]

ER	ercd	Error code

tim detail:

Current time (ms)

ofs detail:

Elapsed time from tim (nanoseconds)

[Error Codes]

E_OK	Normal completion

[Description]

Gets the current time as total elapsed milliseconds since 0:00:00 (GMT), January 1, 1985. The value returned in tim is the same as that obtained by tk_get_tim. tim is expressed in the resolution of timer interrupt intervals (cycles), but even more precise time information is obtained in ofs as the time elapsed from tim in nanoseconds. The resolution of ofs is implementation-dependent, but is generally the timer hardware resolution.

Since tim is time counted based on timer interrupts, in some cases time is not refreshed, when a timer interrupt cycle arrives while interrupts are disabled and the timer interrupt handler is not started (is delayed). In such cases, the time as updated by the previous timer interrupt is returned in tim, and the time elapsed from the previous timer interrupt is returned in ofs. Accordingly, in some cases ofs will be a larger value than the timer interrupt cycle. The length of elapsed time that can be measured by ofs depends on the hardware, but it should preferably be able to measure at least up to twice the timer interrupt cycle (0 <= ofs < twice the timer interrupt cycle).

Note that the time returned in tim and ofs is the time at some point between the calling of and return from td_get_tim. It is neither the time at which td_get_tim was called nor the time of return from td_get_tim. In order to obtain more accurate information, this function should be called in interrupts disabled state.

[Difference with T-Kernel 1.00.00]

The type of ofs is pointer to UW instead of pointer to UINT.

td_get_otm

Get System Operating Time

[C Language Interface]

```
ER ercd = td_get_otm ( SYSTIM *tim, UW *ofs ) ;
```

[Parameters]

SYSTIM*	tim	Address of packet for returning operating time (ms)
UW*	ofs	Location for returning elapsed time from tim (nanoseconds)

[Return Parameters]

ER	ercd	Error code

tim detail:

Operating time (ms)

ofs detail:

Elapsed time from tim (nanoseconds)

[Error Codes]

E_OK	Normal completion

[Description]

Gets the system operating time (uptime, as elapsed milliseconds since the system was booted). The value returned in tim is the same as that obtained by tk_get_otm. tim is expressed in the resolution of timer interrupt intervals (cycles), but even more precise time information is obtained in ofs as the time elapsed from tim in nanoseconds. The resolution of ofs is implementation-dependent, but is generally the timer hardware resolution.

Since tim is time counted based on timer interrupts, in some cases time is not refreshed when a timer interrupt cycle arrives while interrupts are disabled and the timer interrupt handler is not started (is delayed). In such cases, the time as updated by the previous timer interrupt is returned in tim, and the time elapsed from the previous timer interrupt is returned in ofs. Accordingly, in some cases, ofs will be a larger value than the timer interrupt cycle. The length of elapsed time that can be measured by ofs depends on the hardware but it should preferably be able to measure at least up to twice the timer interrupt cycle (0 <= ofs < twice the timer interrupt cycle).

Note that the time returned in tim and ofs is the time at some point between the calling of and return from td_get_otm. It is neither the time at which td_get_otm was called nor the time of return from td_get_otm. In order to obtain more accurate information, this function should be called in interrupts disabled state.

[Difference with T-Kernel 1.00.00]

The type of ofs is pointer to UW instead of pointer to UINT.

td_ref_dsname

Refer to DS Object Name

[C Language Interface]

```
ER ercd = td_ref_dsname ( UINT type, ID id, UB *dsname ) ;
```

[Parameters]

UINT	type	object type
ID	id	object ID
UB	*dsname	address to return DS object name

[Return Parameters]

ER	ercd	Error code
	dsname	DS object name, set at object creation or by td_set_dsname()

[Error Codes]

E_OK	Normal completion
E_PAR	Invalid object type
E_NOEXS	Object does not exist
E_OBJ	DS object name is not used

[Description]

Get DS object name (dsname), which is set at object creation. The object is specified by object type (type) and object ID (id). Object types (type) are as follows:

```
TN_TSK  0x01  /* task */
TN_SEM  0x02  /* semaphore */
TN_FLG  0x03  /* event flag */
TN_MBX  0x04  /* mail box */
TN_MBF  0x05  /* message buffer */
TN_POR  0x06  /* rendezvous port */
TN_MTX  0x07  /* mutex */
TN_MPL  0x08  /* variable-size memory pool */
TN_MPF  0x09  /* fixed-size memory pool */
TN_CYC  0x0a  /* cyclic handler */
TN_ALM  0x0b  /* alarm handler */
```

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() refers to the new name. DS object name needs to satisfy the following conditions, but character code range is not checked by µT-Kernel.

character (UB) range:	a - z, A - Z, 0 – 9
name length:	8 byte (NULL is filled if name length is less than 8 byte)

td_set_dsname

Set DS Object Name

[C Language Interface]

```
ER ercd = td_set_dsname ( UINT type, ID id, UB *dsname ) ;
```

[Parameters]

UINT	type	object type
ID	id	object ID
UB	*dsname	DS object name to be set

[Return Parameters]

ER	ercd	Error code

[Error Codes]

E_OK	Normal completion
E_PAR	Invalid object type
E_NOEXS	Object does not exist
E_OBJ	DS object name is not used

[Description]

Update DS object name (dsname), which is set at object creation. The object is specified by object type (type) and object ID (id). Object types (type) are the same as that for td_ref_dsname(). The DS object name needs to satisfy the following conditions but character code range is not checked by µT-Kernel.

character (UB) range:	a - z, A - Z, 0 - 9
name length:	8 byte (NULL is filled if name length is less than 8 byte)

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.

4.12.2 Trace Functions

These functions enable a debugger to trace program execution. Execution trace is performed by setting hook routines.

- Return from a hook routine must be done after states have returned to where they were when the hook routine was called. Restoring of registers, however, can be done in accordance with the register saving rules of C language functions.
- In a hook routine, limitations on states must not be modified to make them less restrictive than when the routine was called. For example, if the hook routine was called during interrupts disabled state, interrupts must not be enabled.
- A hook routine inherits the stack at the time of the hook. Too much stack use may therefore cause a stack overflow. The extent to which the stack can be used is not definite since it differs with the situation at the time of the hook. Switching to a separate stack in the hook routine would be safer.

td_hok_svc

Define System Call/Extended SVC Hook Routine

[C Language Interface]

```
ER ercd = td_hok_svc ( TD_HSVC *hsvc ) ;
```

[Parameters]

```
TD_HSVC     hsvc        Hook routine definition information
hsvc detail:
    FP      enter       Hook routine before calling the service call
    FP      leave       Hook routine after calling the service call
```

[Return Parameters]

```
ER          ercd        Error code
```

[Description]

Sets hook routines before and after the issuing of a system call or extended SVC. Setting NULL in `hsvc` cancels a hook routine.

The objects of a trace are µT-Kernel system calls (`tk_xxx`) and extended SVC. Depending on the implementation, generally `tk_ret_int` is not the object of a trace.

Debugger support function system calls (`td_xxx`) are not objects of a trace.

A hook routine is called in the context from which the system call or extended SVC was called. For example, the invoking task in a hook routine is the same as the task that invoked the system call or extended SVC.

Since task dispatching and interrupts can occur inside system call processing, `enter`() and `leave`() are not necessarily called in succession as a pair in every case. If a system call is one that does not return, `leave`() will not be called.

```
VP enter( FN fncd, TD_CALINF *calinf, ... )
```

`fncd`	Function code
	< 0 System call
	≥ 0 Extended SVC
`calinf`	Caller information
...	Parameters (variable number)
return code	Any value to be passed to `leave`()

```
typedef struct td_calinf {
```

As information for determining the address from which a system call or extended SVC was called, it is preferable to include information for performing a stack back-trace. The contents are implementation-dependent but generally consist of register values such as stack pointer and program counter.

```
} TD_CALINF;
```

This is called right before a system call or extended SVC. The value passed in the return code is passed on to the corresponding leave(). This makes it possible to confirm the pairing of `enter`() and `leave`() calls or to pass any other information.

```
exinf = enter(fncd, &calinf, ... )
ret = system call or extended SVC execution
leave(fncd , ret, exinf)
```

- System call: the parameters are the same as the system call parameters.

Example:

For system call `tk_wai_sem`(ID semid, INT cnt, TMO `tmout`)

```
enter(TFN_WAI_SEM, &calinf, semid, cnt, tmout)
```

- Extended SVC: the parameters are as in the packet passed to the extended SVC handler. `fncd` is the same as that passed to the extended SVC handler.

```
enter( FN fncd, TD_CALINF *calinf, VP pk_para )
```

`void leave( FN fncd, INT ret, VP exinf )`

`fncd`	Function code
`ret`	Return code of the system call or extended SVC
`exinf`	Any value returned by `enter`()

This is called right after returning from a system call or extended SVC.

When a hook routine is set after a system call or extended SVC is called (while the system call or extended SVC is executing), in some cases, only `leave`() may be called without calling `enter`(). In such a case, NULL is passed in `exinf`. If, on the other hand, a hook routine is canceled after a system call or extended SVC is called, there may be cases when `enter`() is called but not `leave`().

td_hok_dsp

Define Task Dispatch Hook Routine

[C Language Interface]

```
ER ercd = td_hok_dsp ( TD_HDSP *hdsp ) ;
```

[Parameters]

```
TD_HDSP    hdsp       Hook routine definition information
hdsp detail:
    FP     exec       Hook routine when execution starts
    FP     stop       Hook routine when execution stops
```

[Return Parameters]

```
ER         ercd       Error code
```

[Description]

Sets hook routines in the task dispatcher. A hook routine is canceled by setting NULL in `hdsp`.

The hook routines are called in dispatch disabled state. The hook routines must not invoke µT-Kernel system calls (`tk_xxx`) or extended SVC. Debugger support function system calls (`td_xxx`) may be invoked.

```
void exec( ID tskid, INT lsid )
```

`tskid`	Task ID of the started or resumed task
`lsid`	Implementation-defined

This is called when the designated task starts execution or resumes. At the time `exec`() is called, the task designated in `tskid` is already in RUN state and logical space has been switched. However, execution of the `tskid` task program code occurs after the return from `exec`().

```
void stop( ID tskid, INT lsid, UINT tskstat )
```

`tskid`	Task ID of the task stopping execution
`lsid`	Implementation-defined
`tskstat`	State of the task designated in `tskid`

This is called when the designated task executes or stops execution. `tskstat` indicates the task state after stopping as one of the following states.

TTS_RDY	READY state
TTS_WAI	WAIT state
TTS_SUS	SUSPEND state
TTS_WAS	WAIT-SUSPEND state
TTS_DM	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.

[Difference with T-Kernel]

The argument lsid passed to the hook routine is implementation-defined. This is because in µT-Kernel, there is no logical space.

td_hok_int

Define Interrupt Handler Hook Routine

[C Language Interface]

```
ER ercd = td_hok_int ( TD_HINT *hint ) ;
```

[Parameters]

TD_HINT	hint	Hook routine definition information

hint detail:

FP	enter	Hook routine before calling the handler
FP	leave	Hook routine after calling the handler

[Return Parameters]

ER	ercd	Error code

[Description]

Sets hook routines before and after an interrupt handler is called. Hook routine setting cannot be done independently for different exception or interrupt factors. One pair of hook routines is set in common for all exception and interrupt factors.

Setting `hint` to NULL cancels the hook routines.

The hook routines are called as task-independent portion (part of the interrupt handler). Accordingly, the hook routines can call only those system calls that can be invoked from a task-independent portion.

Note that hook routines can be set only for interrupt handlers defined by `tk_def_int` with the `TA_HLNG` attribute. A `TA_ASM` attribute interrupt handler cannot be hooked by a hook routine. Hooking of a `TA_ASM` attribute interrupt handler is possible only by directly manipulating the exception/interrupt vector table. The actual methods are implementation-dependent.

```
void enter( UINT dintno )
void leave( UINT dintno )
```

`dintno` Interrupt definition number

The parameters passed to `enter`() and `leave`() are the same as those passed to the exception/interrupt handler. Depending on the implementation, information other than `dintno` may also be passed. A hook routine is called as follows from a high-level language support routine.

```
enter(dintno);
inthdr(dintno); /* exception/interrupt handler */
leave(dintno);
```

`enter()` is called in interrupts disabled state, and interrupts must not be enabled. Since `leave()` assumes the status on return from `inthdr()`, the interrupts disabled or enabled status is indeterminate.

`enter()` can obtain only the same information as that obtainable by `inthdr()`. Information that cannot be obtained by `inthdr()` cannot be obtained by `enter()`. The information that can be obtained by `enter()`

and inthdr() is guaranteed by the specification to include dintno, but other information is implementation-dependent. Note that since interrupts disabled state and other states may change while leave() is running, leave() does not necessarily obtain the same information as that obtained by enter() or inthdr().

Chapter 5

Reference

5.1 List of C Language Interface µT-Kernel

Task Management Functions

```
ID     tskid = tk_cre_tsk ( 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            tk_exd_tsk ( );
ER      ercd = tk_ter_tsk ( ID tskid );
ER      ercd = tk_chg_pri ( ID tskid, PRI tskpri );
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, T_REGS *pk_regs, T_EIT *pk_eit, T_CREGS *pk_cregs );
ER      ercd = tk_ref_tsk ( ID tskid, T_RTSK *pk_rtsk );
```

Task-Dependent Synchronization Functions

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

Synchronization and Management Functions

```
ID     semid = tk_cre_sem ( 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_ref_sem ( ID semid, T_RSEM *pk_rsem );
ID     flgid = tk_cre_flg ( 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_ref_flg ( ID flgid, T_RFLG *pk_rflg );
ID     mbxid = tk_cre_mbx ( 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_ref_mbx ( ID mbxid, T_RMBX *pk_rmbx );
```

Extended Synchronization and Communication Functions

```
ID     mtxid = tk_cre_mtx ( T_CMTX *pk_cmtx );
ER      ercd = tk_del_mtx ( ID mtxid );
ER      ercd = tk_loc_mtx ( ID mtxid, TMO tmout );
ER      ercd = tk_unl_mtx ( ID mtxid );
ER      ercd = tk_ref_mtx ( ID mtxid, T_RMTX *pk_rmtx );
ID     mbfid = tk_cre_mbf ( T_CMBF *pk_cmbf );
ER      ercd = tk_del_mbf ( ID mbfid );
ER      ercd = tk_snd_mbf ( ID mbfid, VP msg, INT msgsz, TMO tmout );
INT    msgsz = tk_rcv_mbf ( ID mbfid, VP msg, TMO tmout );
ER      ercd = tk_ref_mbf ( ID mbfid, T_RMBF *pk_rmbf );
ID     porid = tk_cre_por ( T_CPOR *pk_cpor );
ER      ercd = tk_del_por ( ID porid );
INT  rmsgsz = tk_cal_por ( ID porid, UINT calptn, VP msg, INT cmsgsz, TMO tmout );
INT  cmsgsz = tk_acp_por ( ID porid, UINT acpptn, RNO *p_rdvno, VP msg, TMO tmout );
ER      ercd = tk_fwd_por ( ID porid, UINT calptn, RNO rdvno, VP msg, INT cmsgsz );
ER      ercd = tk_rpl_rdv ( RNO rdvno, VP msg, INT rmsgsz );
ER      ercd = tk_ref_por ( ID porid, T_RPOR *pk_rpor );
```

Memory Pool Management Functions

```
ID     mpfid = tk_cre_mpf ( T_CMPF *pk_cmpf );
ER      ercd = tk_del_mpf ( ID mpfid );
ER      ercd = tk_get_mpf ( ID mpfid, VP *p_blf, TMO tmout );
ER      ercd = tk_rel_mpf ( ID mpfid, VP blf );
ER      ercd = tk_ref_mpf ( ID mpfid, T_RMPF *pk_rmpf );
ID     mplid = tk_cre_mpl ( T_CMPL *pk_cmpl );
ER      ercd = tk_del_mpl ( ID mplid );
ER      ercd = tk_get_mpl ( ID mplid, W blksz, VP *p_blk, TMO tmout );
ER      ercd = tk_rel_mpl ( ID mplid, VP blk );
ER      ercd = tk_ref_mpl ( ID mplid, T_RMPL *pk_rmpl );
```

Time Management Functions

```
ER      ercd = tk_set_tim ( SYSTIM *pk_tim );
ER      ercd = tk_get_tim ( SYSTIM *pk_tim );
ER      ercd = tk_get_otm ( SYSTIM *pk_tim );
ID     cycid = tk_cre_cyc ( T_CCYC *pk_ccyc );
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 );
ID     almid = tk_cre_alm ( T_CALM *pk_calm );
ER      ercd = tk_del_alm ( ID almid );
ER      ercd = tk_sta_alm ( ID almid, RELTIM almtim );
ER      ercd = tk_stp_alm ( ID almid );
ER      ercd = tk_ref_alm ( ID almid, T_RALM *pk_ralm );
```

Interrupt Management Functions

```
ER      ercd = tk_def_int ( UINT dintno, T_DINT *pk_dint );
void           tk_ret_int ( );

               DI ( UINT intsts );
               EI ( UINT intsts );
BOOL         isDI ( UINT intsts );
```

System Management Functions

```
ER      ercd = tk_rot_rdq ( PRI tskpri );
ID     tskid = tk_get_tid ( );
ER      ercd = tk_dis_dsp ( );
ER      ercd = tk_ena_dsp ( );
ER      ercd = tk_ref_sys ( T_RSYS *pk_rsys );
ER      ercd = tk_ref_ver ( T_RVER *pk_rver );
```

Subsystem Management Functions

```
ER      ercd = tk_def_ssy ( ID ssid, T_DSSY *pk_dssy );
ER      ercd = tk_ref_ssy ( ID ssid, T_RSSY *pk_rssy );
```

Device Management Functions

```
ID         tk_opn_dev  ( UB *devnm, UINT omode );
ER         tk_cls_dev  ( ID dd, UINT option );
ID         tk_rea_dev  ( ID dd, W start, VP buf, W size, TMO tmout );
ER        tk_srea_dev  ( ID dd, W start, VP buf, W size, W *asize );
ID         tk_wri_dev  ( ID dd, W start, VP buf, W size, TMO tmout );
ER        tk_swri_dev  ( ID dd, W start, VP buf, W size, W *asize );
ID         tk_wai_dev  ( ID dd, ID reqid, W *asize, ER *ioer, TMO tmout );
INT        tk_sus_dev  ( UINT mode );
ID         tk_get_dev  ( ID devid, UB *devnm );
ID         tk_ref_dev  ( UB *devnm, T_RDEV *pk_rdev );
ID        tk_oref_dev  ( ID dd, T_RDEV *pk_rdev );
INT        tk_lst_dev  ( T_LDEV *pk_ldev, INT start, INT ndev );
INT        tk_evt_dev  ( ID devid, INT evttyp, VP evtinf );
ID         tk_def_dev  ( UB *devnm, T_DDEV *pk_ddev, T_IDEV *pk_idev );
ERT_IDEV   tk_ref_idv  ( T_IDEV *pk_idev  );
```

Debugger Support Functions

```
INT       ct = tk_cre_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_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_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_alm ( ID almid, TD_RALM *ralm );
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_get_reg ( ID tskid, T_REGS *pk_regs, T_EIT *pk_eit, T_CREGS *pk_cregs );
ER      ercd = td_set_reg ( ID tskid, T_REGS *pk_regs, T_EIT *pk_eit, T_CREGS *pk_cregs );
ER      ercd = td_ref_sys ( TD_RSYS *pk_rsys );
ER      ercd = td_get_tim ( SYSTIM *tim, UW *ofs );
ER      ercd = td_get_otm ( SYSTIM *tim, UW *ofs );
ER      ercd = td_ref_dsname ( UINT type, ID id, UB *dsname );
ER      ercd = td_set_dsname ( UINT type, ID id, UB *dsname );
```

Trace Functions

```
    ER      ercd = td_hok_svc ( TD_HSVC *hsvc );
    ER      ercd = td_hok_dsp ( TD_HDSP *hdsp );
    ER      ercd = td_hok_int ( TD_HINT *hint );
```

5.2 List of Error Codes

Normal Completion Error Class (0)

E_OK 0 Normal completion

Internal Error Class (5 to 8)

E_SYS -5 System error

An error of unknown cause affecting the system as a whole.

E_NOCOP -6 The specified coprocessor cannot be used

Not used in µT-Kernel. This error code exists to ensure compatibility with T-Kernel.

Unsupported Error Class (9 to 16)

E_NOSPT -9 Unsupported function

When some system call functions are not supported and such a function was specified, error code E_RSATR or E_NOSPT is returned. If E_RSATR does not apply, error code E_NOSPT is returned.

E_RSFN -10 Reserved function code number

This error code is returned when it is attempted to execute a system call specifying a reserved function code (undefined function code), and also when it is attempted to execute an undefined extended SVC handler.

E_RSATR -11 Reserved attribute

This error code is returned when an undefined or unsupported object attribute is specified.

Checking for this error may be omitted if system-dependent optimization is implemented.

Parameter Error Class (17 to 24)

E_PAR -17 Parameter error

Checking for this error may be omitted if system-dependent optimization is implemented.

E_ID -18 Invalid ID number

E_ID is an error that occurs only for objects having an ID number. Error code E_PAR is returned when a static error is detected in the parameter, such as reserved number or out of range for parameters such as interrupt definition numbers.

Call Context Error Class (25 to 32)

E_CTX -25 Context error

This error indicates that the specified system call cannot be issued in the current context (task portion/task-independent portion or handler RUN state).

This error must be issued whenever there is a meaningful context error in issuing a system call, such as calling from a task-independent portion a system call that may put the invoking task in WAIT state. Due to implementation limitations, there may be other system calls that when called from a given context (such as an interrupt handler) will cause this error to be returned.

E_MACV -26	Memory cannot be accessed; memory access privilege error

Error detection is implementation-dependent.

E_OACV -27	Object Access privilege error

The error indicates that an object inaccessible from the context which called the system call is specified was accessed. This error code is returned when a user task tries to manipulate a system object.

System object definition and error detection are implementation-defined.

E_ILUSE -28	System call illegal use

Resource Constraint Error Class (33 to 40)

E_NOMEM -33	Insufficient memory

This error code is returned when there is insufficient memory (no memory) for allocating an object control block space, user stack space, memory pool space, message buffer space, etc.

E_LIMIT -34	System limit exceeded

This error code is returned when it is attempted to create more of an object than the system allows.

Object State Error Class (41 to 48)

E_OBJ -41	Invalid object state
E_NOEXS -42	Object does not exist
E_QOVR -43	Queuing or nesting overflow

Wait Error Class (49 to 56)

E_RLWAI -49	WAIT state released
E_TMOUT -50	Polling failed or timeout
E_DLT -51	The object being waited for was deleted
E_DISWAI -52	Wait released by wait disabled state

Not used in µT-Kernel. This error code exists to ensure compatibility with T-Kernel

Device Error Class (57 to 64)

E_IO -57	IO error
E_NOMDA -58	No media

Status Error Class (65 to 72)

E_BUSY -65	Busy
E_ABORT -66	Processing was aborted
E_RONLY -67	Write protected