

IEEE Standard for a Real-Time Operating System (RTOS) for Small-Scale Embedded Systems

IEEE Consumer Electronics Society

Sponsored by the
Standards Committee

IEEE
3 Park Avenue
New York, NY 10016-5997
USA

IEEE Std 2050™-2018

IEEE Standard for a Real-Time Operating System (RTOS) for Small-Scale Embedded Systems

Sponsor

Standards Committee
of the
IEEE Consumer Electronics Society

Approved 11 May 2018

IEEE-SA Standards Board

Abstract: A real-time operating system (RTOS) called *μT-Kernel* for small-scale embedded systems such as systems with a single chip microcomputer including 16-bit CPUs, systems with a small amount of ROM/RAM, and systems without a memory management unit (MMU) are specified in this standard.

Keywords: 16-bit CPU, alarm handler, API, Application Programming Interface, cyclic handler, device management, embedded, event flag, fast lock, fast multi-lock, IEEE 2050™, interrupt handler, kernel, IoT edgenode, mailbox, memory pool, message buffer, mutex, non-task portion, physical timer, power management, power saving, priority-based, quasi-task portion, real time, real-time operating system (RTOS), semaphore, service profile, single chip microcomputer, single chip microcontroller, small scale, system call, task, task dispatching, task portion, task-independent portion, task scheduling, TRON, *μT-Kernel*

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

Copyright © 2018 by The Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 24 August 2018. Printed in the United States of America.

IEEE is a registered trademark in the U.S. Patent & Trademark Office, owned by The Institute of Electrical and Electronics Engineers, Incorporated.

PDF: ISBN 978-1-5044-4858-1 STD23105
Print: ISBN 978-1-5044-4859-8 STDPD23105

IEEE prohibits discrimination, harassment, and bullying.

For more information, visit <http://www.ieee.org/web/aboutus/whatis/policies/p9-26.html>.

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

Important Notices and Disclaimers Concerning IEEE Standards Documents

IEEE documents are made available for use subject to important notices and legal disclaimers. These notices and disclaimers, or a reference to this page, appear in all standards and may be found under the heading “Important Notices and Disclaimers Concerning IEEE Standards Documents.” They can also be obtained on request from IEEE or viewed at <http://standards.ieee.org/IPR/disclaimers.html>.

Notice and Disclaimer of Liability Concerning the Use of IEEE Standards Documents

IEEE Standards documents (standards, recommended practices, and guides), both full-use and trial-use, are developed within IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (“IEEE-SA”) Standards Board. IEEE (“the Institute”) develops its standards through a consensus development process, approved by the American National Standards Institute (“ANSI”), which brings together volunteers representing varied viewpoints and interests to achieve the final product. IEEE Standards are documents developed through scientific, academic, and industry-based technical working groups. Volunteers in IEEE working groups are not necessarily members of the Institute and participate without compensation from IEEE. While IEEE administers the process and establishes rules to promote fairness in the consensus development process, IEEE does not independently evaluate, test, or verify the accuracy of any of the information or the soundness of any judgments contained in its standards.

IEEE Standards do not guarantee or ensure safety, security, health, or environmental protection, or ensure against interference with or from other devices or networks. Implementers and users of IEEE Standards documents are responsible for determining and complying with all appropriate safety, security, environmental, health, and interference protection practices and all applicable laws and regulations.

IEEE does not warrant or represent the accuracy or content of the material contained in its standards, and expressly disclaims all warranties (express, implied and statutory) not included in this or any other document relating to the standard, including, but not limited to, the warranties of: merchantability; fitness for a particular purpose; non-infringement; and quality, accuracy, effectiveness, currency, or completeness of material. In addition, IEEE disclaims any and all conditions relating to: results; and workmanlike effort. IEEE standards documents are supplied “AS IS” and “WITH ALL FAULTS.”

Use of an IEEE standard is wholly voluntary. The existence of an IEEE standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard.

In publishing and making its standards available, IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity nor is IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing any IEEE Standards document, should rely upon his or her own independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

IN NO EVENT SHALL IEEE BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO: PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE PUBLICATION, USE OF, OR RELIANCE UPON ANY STANDARD, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE AND REGARDLESS OF WHETHER SUCH DAMAGE WAS FORESEEABLE.

Translations

The IEEE consensus development process involves the review of documents in English only. In the event that an IEEE standard is translated, only the English version published by IEEE should be considered the approved IEEE standard.

Official statements

A statement, written or oral, that is not processed in accordance with the IEEE-SA Standards Board Operations Manual shall not be considered or inferred to be the official position of IEEE or any of its committees and shall not be considered to be, or be relied upon as, a formal position of IEEE. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position of IEEE.

Comments on standards

Comments for revision of IEEE Standards documents are welcome from any interested party, regardless of membership affiliation with IEEE. However, IEEE does not provide consulting information or advice pertaining to IEEE Standards documents. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Since IEEE standards represent a consensus of concerned interests, it is important that any responses to comments and questions also receive the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to comments or questions except in those cases where the matter has previously been addressed. For the same reason, IEEE does not respond to interpretation requests. Any person who would like to participate in revisions to an IEEE standard is welcome to join the relevant IEEE working group.

Comments on standards should be submitted to the following address:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
Piscataway, NJ 08854 USA

Laws and regulations

Users of IEEE Standards documents should consult all applicable laws and regulations. Compliance with the provisions of any IEEE Standards document does not imply compliance to any applicable regulatory requirements. Implementers of the standard are responsible for observing or referring to the applicable regulatory requirements. IEEE does not, by the publication of its standards, intend to urge action that is not in compliance with applicable laws, and these documents may not be construed as doing so.

Copyrights

IEEE draft and approved standards are copyrighted by IEEE under U.S. and international copyright laws. They are made available by IEEE and are adopted for a wide variety of both public and private uses. These include both use, by reference, in laws and regulations, and use in private self-regulation, standardization, and the promotion of engineering practices and methods. By making these documents available for use and adoption by public authorities and private users, IEEE does not waive any rights in copyright to the documents.

Photocopies

Subject to payment of the appropriate fee, IEEE will grant users a limited, non-exclusive license to photocopy portions of any individual standard for company or organizational internal use or individual, non-commercial use only. To arrange for payment of licensing fees, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Updating of IEEE Standards documents

Users of IEEE Standards documents should be aware that these documents may be superseded at any time by the issuance of new editions or may be amended from time to time through the issuance of amendments, corrigenda, or errata. A current IEEE document at any point in time consists of the current edition of the document together with any amendments, corrigenda, or errata then in effect.

Every IEEE standard is subjected to review at least every ten years. When a document is more than ten years old and has not undergone a revision process, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE standard.

In order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit the IEEE Xplore at <http://ieeexplore.ieee.org/> or contact IEEE at the address listed previously. For more information about the IEEE-SA or IEEE's standards development process, visit the IEEE-SA Website at <http://standards.ieee.org>.

Errata

Errata, if any, for all IEEE standards can be accessed on the IEEE-SA Website at the following URL: <http://standards.ieee.org/findstds/errata/index.html>. Users are encouraged to check this URL for errata periodically.

Patents

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken by the IEEE with respect to the existence or validity of any patent rights in connection therewith. If a patent holder or patent applicant has filed a statement of assurance via an Accepted Letter of Assurance, then the statement is listed on the IEEE-SA Website at <http://standards.ieee.org/about/sasb/patcom/patents.html>. Letters of Assurance may indicate whether the Submitter is willing or unwilling to grant licenses under patent rights without compensation or under reasonable rates, with reasonable terms and conditions that are demonstrably free of any unfair discrimination to applicants desiring to obtain such licenses.

Essential Patent Claims may exist for which a Letter of Assurance has not been received. The IEEE is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims, or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

Participants

At the time this IEEE standard was completed, the Internet of Things Working Group had the following membership:

Stephen Dukes, *Chair*
Akira Matsui, *Vice Chair*
Chiaki Ishikawa, *Secretary*

Mashiro Bessho
Atsushi Hasegawa
Hiroki Hihara

Nobushige Hirosato
Tomomi Ishikura

Masakazu Kobayashi
Yutaka Tamanoi
Takeshi Yashiro

The following members of the entity balloting committee voted on this standard. Balloters may have voted for approval, disapproval, or abstention.

Hitachi, Ltd.
Motiveware Technology Co. Ltd.
NEC Corporation

Personal Media Corporation
Renesas Electronics Corporation
TOSHIBA Corporation
Toyo University

TRON Forum
Yokosuka Telecom Research
Park, Inc.

When the IEEE-SA Standards Board approved this standard on 11 May 2018, it had the following membership:

Jean-Philippe Faure, *Chair*
Gary Hoffman, *Vice Chair*
John D. Kulick, *Past Chair*
Konstantinos Karachalios, *Secretary*

Ted Burse
Guido R. Hiertz
Christel Hunter
Joseph L. Koepfinger*
Thomas Koshy
Hung Ling
Dong Liu

Xiaohui Liu
Kevin Lu
Daleep Mohla
Andrew Myles
Paul Nikolich
Ronald C. Petersen
Annette D. Reilly

Robby Robson
Dorothy Stanley
Mehmet Ulema
Phil Wennblom
Philip Winston
Howard Wolfman
Jingyi Zhou

*Member Emeritus

Introduction

This introduction is not part of IEEE Std 2050™-2018, IEEE Standard for a Real-Time Operating System (RTOS) for Small-Scale Embedded Systems.

This standard defines the specification of a real-time operating system (RTOS) called μ T-Kernel. μ T-Kernel is the latest result from the TRON project, which was started by Dr. Ken Sakamura, then at the University of Tokyo in 1984.¹

The TRON Project envisioned that environments optimized to humans would be created by embedding small microprocessors, invented in the prior decade, in many objects in our surroundings and having them talk to each other. In the TRON Project, the computing paradigm to achieve this goal was called a “Highly Functionally Distributed System (HFDS)” and an RTOS called ITRON was created to control such microprocessors efficiently. The specification of the first version of ITRON, namely ITRON1, was published in 1987. The project promoted the industry-academic cooperation and published the technical specification and other information so that anyone can make use of the technology for free under the philosophy of “Open Approach.” As a result, ITRON specification OS was born and it ran on many types of processors. It became the de facto standard RTOS for embedded computer systems. Additionally, the development of “ μ ITRON,” which is an improved version of ITRON and has better adaptability, proceeded concurrently.² OSs based on ITRON and μ ITRON specifications have been used widely in many embedded computer systems: they are used in consumer products, such as home electronic appliances and AV equipment, and industrial applications, such as machine control on factory floors, engine control of automobiles, etc.

The concept of “HFDS,” which the TRON Project proposed, started to be called “ubiquitous computing” before the turn of the century and is now widely recognized as the Internet of Things (IoT). Since its inception in 1984, the TRON Project has targeted the IoT in today's parlance as the main application field of microprocessors and carried out research and development of OS and computer architecture. The latest result of such research and development of OS is μ T-Kernel, a resource-efficient RTOS suitable for IoT edge nodes. It is an improvement of μ ITRON, and has features for IoT.

μ T-Kernel improves development efficiency of the software by standardizing basic OS functions and API specification. It is designed to deliver high performance even on a lower-end single-chip microcontroller unit (MCU), including 16-bit MCU, MCUs without a memory management unit (MMU), and small-scale embedded systems with a small amount ROM/RAM. Furthermore, μ T-Kernel has functions such as device driver control and power saving, so it can build low-power systems in which various types of devices and communication methods are embedded for building an IoT network. Today, the TRON Forum, a non-profit organization chaired by Dr. Ken Sakamura, is the main proponent of TRON Project activity.

Based on the adoptions so far, it has been reported that more than half of embedded devices use the results of the TRON Project in the 2010s, 30 years after the inception of the project.³ In the IoT age, the RTOS family developed by the TRON Project is the most suitable basic technology to build high-performance IoT edge node devices, which operate in an efficient manner with small resources and small power consumption. With this background, the TRON Project has decided to standardize the specification of μ T-Kernel, its latest version of RTOS, as IEEE Std 2050™-2018, to further improve the development efficiency of IoT edge nodes by making the specification widely recognized as an IEEE standard.

¹ Information is available at <http://www.tron.org/>.

² See μ ITRON 3.0: An Open and Portable Real-Time Operating System for Embedded Systems : Concept and Specification, Ken Sakamura, IEEE Computer Society Press Los Alamitos, CA, USA ©1997, ISBN: 0818677953.

³ Information is available at <http://www.tron.org/blog/2017/07/press20170406/>.

Contents

1. Overview	1
1.1 Scope	1
1.2 Positioning and basic design policy of μ T-Kernel	1
1.3 Structure	2
1.4 Implementation specification document	2
2. Definitions	3
3. μ T-Kernel concepts	4
3.1 Basic terminology	4
3.2 Task states and scheduling rules	6
3.3 Interrupt handling	11
3.4 Task exception handling	11
3.5 System states	12
3.6 Objects	14
3.7 Protection levels	15
3.8 Service profile	16
4. Common rules of μ T-Kernel	17
4.1 Data types	17
4.2 System calls	19
4.3 High-level language support routines	25
4.4 Service profile	26
4.5 API notation	32
5. μ T-Kernel/OS functions	34
5.1 Task management functions	34
5.2 Task synchronization functions	54
5.3 Task exception handling functions	75
5.4 Synchronization and communication functions	85
5.5 Extended synchronization and communication functions	114
5.6 Memory pool management functions	136
5.7 Time management functions	152
5.8 Interrupt management functions	177
5.9 System management functions	183
6. μ T-Kernel/SM functions	194
6.1 System memory management functions	194
6.2 Device management functions	199
6.3 Interrupt management functions	252
6.4 I/O port access support functions	265
6.5 Power management functions	274
6.6 System configuration information management functions	277
6.7 Memory cache control functions	282
6.8 Physical timer functions	286
6.9 Utility functions	295

Annex A (informative) System configuration	306
Annex B (informative) List of C language references.....	307
B.1 μ T-Kernel/OS	307
B.2 μ T-Kernel/SM.....	310
Annex C (informative) List of error codes	314
C.1 Normal completion error class (0)	314
C.2 Normal completion internal error class (5 to 8)	314
C.3 Unsupported error class (9 to 16).....	314
C.4 Parameter error class (17 to 24)	315
C.5 Call context error class (25 to 32).....	315
C.6 Resource constraint error class (33 to 40).....	315
C.7 Object state error class (41 to 48)	316
C.8 Wait error class (49 to 56).....	316
C.9 Device error class (57 to 64) (μ T-Kernel/SM).....	316
C.10 Status error class (65 to 72) (μ T-Kernel/SM)	316
Annex D (informative) List of APIs and service profile names	317
D.1 μ T-Kernel/OS	317
D.2 μ T-Kernel/SM	320

IEEE Standard for a Real-Time Operating System (RTOS) for Small-Scale Embedded Systems

1. Overview

1.1 Scope

This standard is a real-time operating system (RTOS) specification for small-scale embedded systems such as systems with a single chip microcomputer (single chip microcontroller) including 16-bit CPUs, systems with a small amount of ROM/RAM, and systems without a memory management unit (MMU).

The RTOS defined in this document is called *μT-Kernel*. This RTOS specification includes *μT-Kernel/OS*, which provides the basic functions of RTOS such as scheduling, synchronization, and communication of tasks. The RTOS specification also includes *μT-Kernel/SM (System Manager)*, which provides the extension function for system management.

1.2 Positioning and basic design policy of μT-Kernel

The μT-Kernel specification defines an embedded RTOS with a small resource footprint meant for controlling IoT edge devices. Standardization OS functions and Application Programming Interface (API) improves the reusability and distribution of software and the development of applications. It is also designed so that it can perform well on resource poor systems such those with a single chip microcontroller unit (MCU), or MCUs without MMUs, and systems with a small amount of ROM/RAM. It also features device driver management functions, energy-saving functions, etc., to build resource-efficient systems that support various types of network protocols for the IoT and incorporate various devices.

The μT-Kernel specification defines a standard for an RTOS, and it can be implemented on many types of CPUs irrespective of CPU architectures. At the same time, the designers are aware that it makes sense to adapt the OS implementation or limit the OS functions in the case of very resource-poor systems in order to cope with a particular choice of CPU and hardware configuration as in the case of tiny IoT edge nodes. To cope with such situations, a concept and description of “service profile” has been introduced in μT-Kernel to leave room for the flexible implementation of the OS, at the same time retaining the compatibility of software and portability. Service profile makes it possible to formally describe the omission and/or difference of functions available in a particular implementation of the OS. This makes it easy for middleware and applications that run under the OS to learn and cope with the implementation-dependent differences.

The μ T-Kernel specification includes common definitions such as data types, the state transition of tasks, which are the basic units of parallel execution inside programs, non-task behavior such as that of interrupt handler, and API specifications provided by the OS. File system management, network communication, process management, etc., are not included in the μ T-Kernel specification. However, by adding appropriate middleware packages, we can build relatively large systems using μ T-Kernel. That is, μ T-Kernel can be used as a microkernel for a large system that has functions such as file system management, network communication, and process management. It can be extended to support multi-core processors as well.

1.3 Structure

μ T-Kernel consists of μ T-Kernel/OS that handles the intrinsic functions of RTOS such as task scheduling, synchronization, and inter-/intra-task communication, and μ T-Kernel/SM that offers additional system management functions.

μ T-Kernel/OS provides the following functions:

- Task management functions
- Task synchronization functions
- Task exception handling functions
- Synchronization and communication functions
- Extended synchronization and communication functions
- Memory pool management functions
- Time management functions
- Interrupt management functions

μ T-Kernel/SM provides the following functions:

- Device management functions
- Interrupt management functions
- I/O port access support functions
- Power management functions
- System configuration information management functions
- Memory cache control functions
- Physical timer functions
- Utility functions

1.4 Implementation specification document

With the introduction of a service profile, not all the implementations of μ T-Kernel provide all the features in the specification.

Hence, to let the user verify the implementation-dependent details of a particular implementation of μ T-Kernel, and avoid spending time to understand unexpected behavior, all μ T-Kernel implementations

shall produce an implementation specification document. The implementation specification document shall describe at least the following:

Version number of μ T-Kernel specification

Clearly specify the major and minor version numbers of μ T-Kernel.

Information about the service profile

Explicitly specify the values of all the service profile items.

Information about the system configuration

See Annex A for the system configuration.

2. Definitions

For the purposes of this document, the following terms and definitions apply. The *IEEE Standards Dictionary Online* should be consulted for terms not defined in this clause.¹

API: *See:* application programming interface.

application programming interface (API): The standard interface for calling functions provided by μ T-Kernel from applications or middleware.

context: The execution environment (CPU mode, stack configuration, etc.) in which a program runs.

dispatcher: The kernel mechanism that realizes task dispatching. *Syn:* **task dispatcher.**

dispatching: The switching of tasks executed by the processor. *Syn:* **task dispatching.**

extended supervisor call (SVC): System calls that are added at the time of the initial startup of the real-time operating system (RTOS) or added later.

extended SVC handler: A program that executes the function of an extended SVC.

implementation-defined: Something that is not standardized in the μ T-Kernel specification and must be defined for each implementation.

implementation-dependent: Something whose behavior varies according to the target systems or system operating conditions, and cannot and/or should not be defined.

invoking task: A task that invokes a system call.

kernel: The portion of μ T-Kernel that is not implemented by extended SVCs, compile-time macros, or library functions.

precedence: The order of task execution that is the order relation when the execution right is assigned to a group of ready tasks.

scheduler: The kernel mechanism that realizes scheduling, i.e., task scheduling. *Syn:* **task scheduler.**

scheduling: The processing that determines which task to execute next. *Syn:* **task scheduling.**

¹IEEE Standards Dictionary Online is available at: <http://dictionary.ieee.org>.

supervisor call (SVC): *See: system call.*

system call [supervisor call (SVC)]: Some of the Application Programming Interfaces (APIs) provided by the μ T-Kernel specification that directly calls kernel functions.

task: The basic logical unit of concurrent program execution.

task dispatcher: *See: dispatcher.*

task dispatching: *See: dispatching.*

task scheduler: *See: scheduler.*

task scheduling: *See: scheduling.*

3. μ T-Kernel concepts

3.1 Basic terminology

3.1.1 Real-time system, real-time operating system (RTOS)

A system whose response time and delay time are deterministic without uncertainty and non-reproducibility and has an internal configuration that makes the worst value predictable or makes it easy to produce an educated guess value is called a *real-time system*.

μ T-Kernel is the real-time operating system (RTOS) that is used for building real-time systems with the preceding characteristics.

3.1.2 Task, invoking task

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

A task that invokes a system call is called an *invoking task*.

3.1.3 Dispatch, dispatcher

Switching of tasks executed by the processor is called *dispatching* (or *task dispatching*). The kernel mechanism that realizes dispatching is called a *dispatcher* (or *task dispatcher*).

3.1.4 Scheduling, scheduler

Processing that determines which task to execute next is called *scheduling* (or *task scheduling*). The kernel mechanism that realizes scheduling is called *scheduler* (or *task scheduler*). Generally a task scheduler is implemented inside system call processing or in the dispatcher.

3.1.5 Context

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

3.1.6 Precedence

The execution order of tasks, i.e., the order relation, is called *precedence*. This refers to the order of tasks when an execution right is given to a task among a group of tasks in the executable state to be in the execution state. If task Y has a higher precedence than task X, task Y will be executed first. If task Y, which has higher precedence than task X, becomes ready for execution while task X is executed, the execution right will be transferred to task Y, and task Y will be in execution state, i.e., RUNNING state. In this case, task X will be in the executable state, i.e., READY state, instead of execution state.

NOTE—*Precedence* has a similar meaning to *priority*, and they both affect the execution order of tasks. However, “priority” is an attribute of tasks specified by API parameter, etc., explicitly from applications, whereas “precedence” is a concept that is employed to define the execution order among a group of tasks. The precedence among a group of tasks is determined based on the priority of the tasks. A task with higher priority has higher precedence. On the other hand, tasks with the same priority do not have the same precedence. Among tasks having the same priority, the one that entered an executable state (i.e., RUNNING state or READY state) first has the highest precedence. It is possible, however, to use an API such as `tk_rot_rdq` to change the precedence among tasks having the same priority.

3.1.7 API and system call

The standard interface for calling functions provided by μ T-Kernel from applications or middleware is collectively called *API*. In addition to system calls that directly call kernel functions, APIs include functions implemented as extended SVCs, compile-time macros, and libraries.

3.1.8 Extended SVC

System calls that are added at the time of the initial startup of OS or added later are called *extended SVC*. The μ T-Kernel specification does not define how to add extended SVC (its method and APIs), however, a particular implementation can introduce the function of extended SVC in an implementation-dependent manner. Implementation of μ T-Kernel/SM API can use extended SVC(s).

A program that executes the function of an extended SVC is *extended SVC handler*.

3.1.9 Kernel

Kernel refers to the portion of μ T-Kernel that is not implemented by extended SVCs, compile-time macros, or library functions.

μ T-Kernel/SM API can be implemented using extended SVC, compile-time macros, and/or library functions. Such APIs are part of μ T-Kernel specification. However, it is not deemed to be part of the kernel. On the other hand, all the functions of μ T-Kernel/OS are included in the kernel.

When we refer to system state while a non-task portion is executing, we need to be aware whether the execution is within the kernel or not.

3.1.10 Implementation-defined

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

3.1.11 Implementation-dependent

Implementation-dependent means that in the μ T-Kernel specification, the behavior of something varies according to the target systems or system operating conditions, and cannot and/or should not be specified. The behavior should be defined for each implementation. The specifics of the implementation should be described clearly in the implementation specifications. In application programs, the portion dependent on implementation-dependent items needs to be modified when porting in principle.

3.2 Task states and scheduling rules

3.2.1 Task states

Task states are classified into the following five states. Of these, waiting states are further classified into three states. Saying that a task is in a *runnable state* means it is in either a RUNNING state or a READY state.

3.2.1.1 RUNNING state

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

3.2.1.2 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 a READY state.

3.2.1.3 Waiting states

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

a) WAITING state

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

b) **SUSPENDED state**

Execution was forcibly interrupted by another task.

c) **WAITING-SUSPENDED state**

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

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

3.2.1.4 DORMANT state

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

3.2.1.5 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 preceding categories (see 3.5).

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

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

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

Task state transitions in a typical implementation are shown in Figure 1. Depending on the implementation, there may be other states besides those shown here.

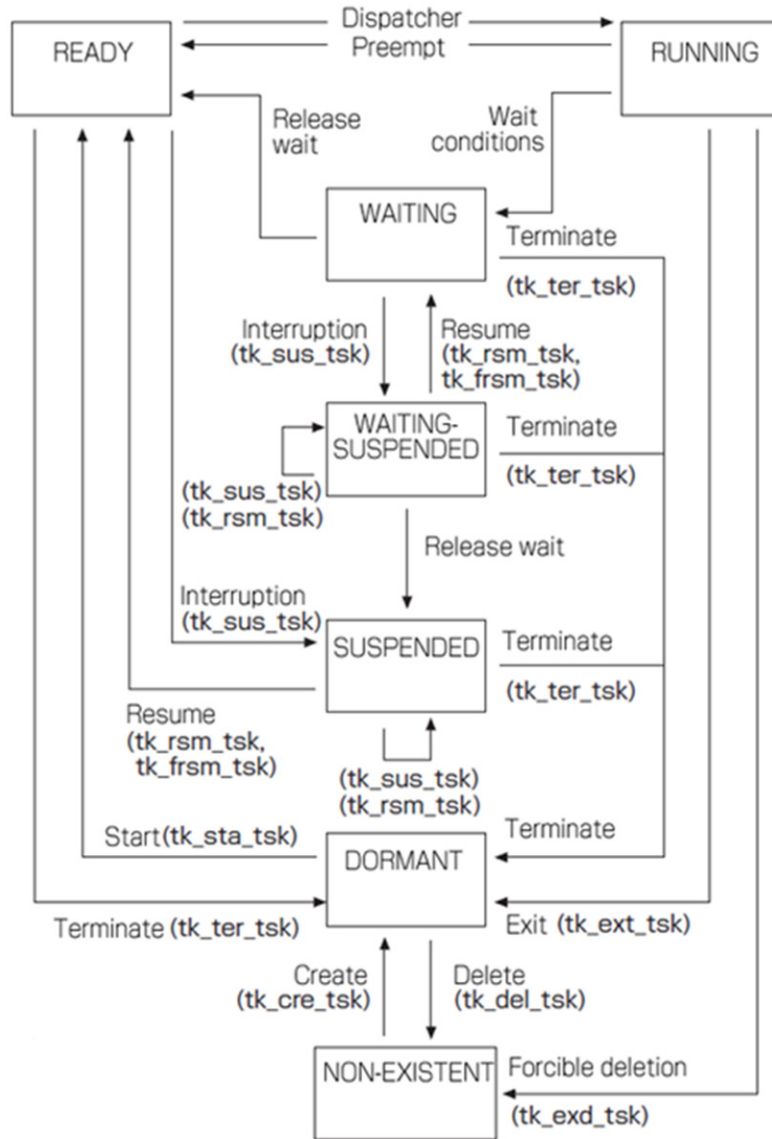


Figure 1—Task state transition diagram

A feature of μ T-Kernel is the clear distinction made between system calls that perform operations affecting the invoking task and those whose operations affect other tasks (see Table 1). The reason for this is to clarify task state transitions and facilitate understanding of system calls. This distinction between system call operations in the invoking task and operations affecting other tasks can also be seen as a distinction between state transitions from RUNNING state and those from other states.

Table 1—State transitions distinguishing invoking task and other tasks

	Operations in invoking tasks (transition from RUNNING state)	Operations on other tasks (transitions from other states)
Task transition to a waiting state (including SUSPENDED)	tk slp tsk RUNNING state → WAITING state	tk sus tsk READY state, WAITING state → SUSPENDED state, WAITING-SUSPENDED state
Task exit	tk ext tsk RUNNING state → DORMANT state	tk ter tsk READY state, WAITING state → DORMANT state
Task deletion	tk exd tsk RUNNING state → NON-EXISTENT state	tk del tsk DORMANT state → NON-EXISTENT state

NOTE 1—WAITING state and SUSPENDED state are orthogonally related, in that a request for transition to SUSPENDED state cannot have any effect on the conditions for task wait release. That is, the task wait release conditions are the same whether the task is in WAITING state or WAITING-SUSPENDED state. Thus, even if transition to SUSPENDED state is requested for a task that is in a state of waiting to acquire some resource (semaphore resource, memory block, etc.), and the task goes to WAITING-SUSPENDED state, the conditions for allocation of the resource do not change but remain the same as before the request to go to SUSPENDED state. ¹

NOTE 2—The reason the μ T-Kernel makes a distinction between WAITING state (wait caused by the invoking task) and SUSPENDED state (wait caused by another task) is that these states sometimes overlap. By recognizing these overlapped states as WAITING-SUSPENDED states, the task state transitions become clearer and system calls are easier to understand. On the other hand, since a task in WAITING state cannot invoke a system call, different types of WAITING state (e.g., waiting for wakeup, or waiting to acquire a semaphore resource) will never overlap. Since there is only one kind of waiting state caused by another task (SUSPENDED state), the μ T-Kernel treats repeated entries to SUSPENDED state as nesting, thereby achieving clarity of task state transitions.

3.2.2 Task scheduling rules

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

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

3.2.2.1 Additional notes and examples for task scheduling rules

According to the scheduling rules adopted in the μ T-Kernel, so long as there is a higher precedence task in runnable states, a task with lower precedence will simply not run. That is, unless the highest-precedence task goes to WAITING state or for other reason cannot run, other tasks are not run. This is a fundamental difference from TSS (time sharing system) scheduling in which multiple tasks are treated equally.

¹ Notes in text, tables, and figures of a standard are given for information only and do not contain requirements needed to implement this standard.

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

Examples in the following figures illustrate how the task that first goes to runnable states (RUNNING state or READY state) gains precedence among tasks having the same priority. Figure 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 RUNNING state.

When Task A exits, Task B with the next-highest precedence goes to RUNNING state (Figure 3). When Task A is again started, Task B is preempted and reverts to READY state; but since Task B went to runnable states earlier than Task C and Task D, it still has the highest precedence among tasks with the same priority. In other words, the task precedence reverts to that in Figure 2.

Next, consider what happens when Task B goes to WAITING state in the conditions in Figure 3. Since task precedence is defined among tasks that can be run, the precedence among tasks becomes as shown in Figure 4. Thereafter when the Task B waiting state is released, Task B goes to runnable states after Task C and Task D, and thus assumes the lowest precedence among tasks of the same priority (Figure 5).

To summarize, immediately after a task that goes from READY state to RUNNING state reverts to READY state, it has the highest precedence among tasks of the same priority; but after a task goes from RUNNING state to WAITING state and then the wait is released, its precedence is the lowest among tasks of the same priority.

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

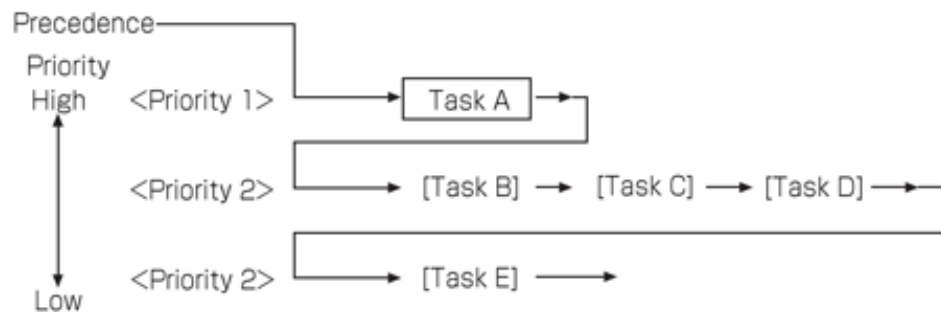


Figure 2—Precedence in initial state

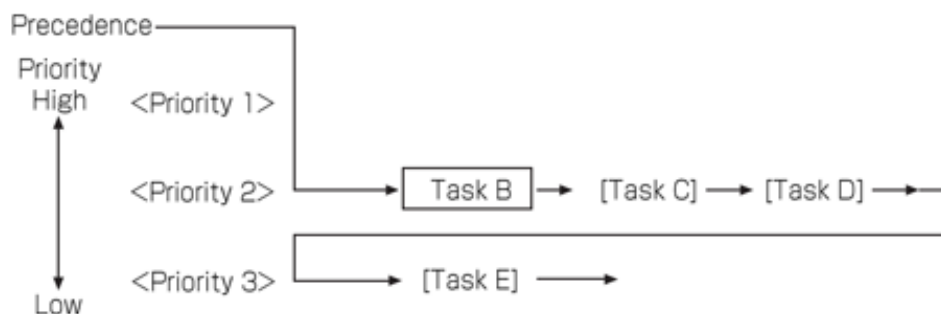


Figure 3—Precedence after Task B goes to RUNNING state

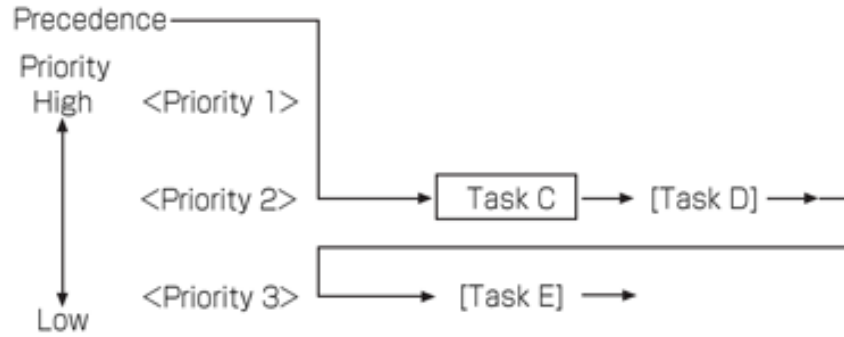


Figure 4—Precedence after Task B goes to WAITING state

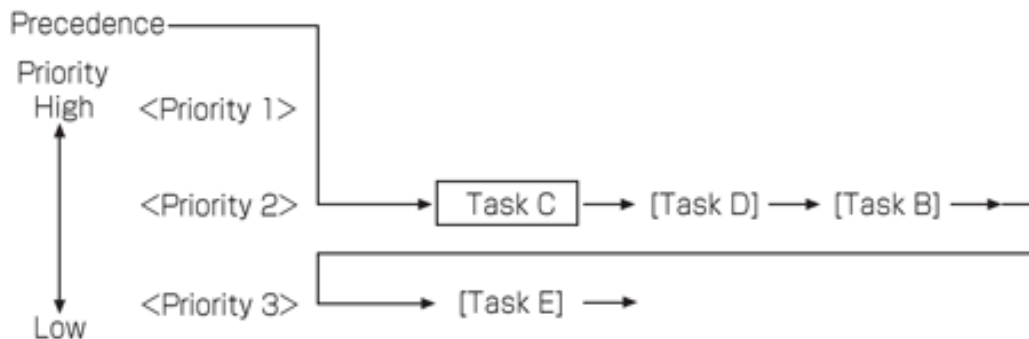


Figure 5—Precedence after Task B WAITING state is released

3.3 Interrupt handling

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

For more details, see 5.8.

3.4 Task exception handling

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

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

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

For more details, see 5.3.

3.5 System states

3.5.1 System states while non-task portion is executing

When programming tasks to run on μ T-Kernel, one can keep track of the changes in task states by using a task state transition diagram. In the case of routines such as interrupt handlers or extended SVC handlers, however, the user must perform programming at a level closer to the kernel than tasks. In such cases consideration must be made also of system states while a non-task portion is being executed, for application programs to work properly. An explanation of μ T-Kernel system states is therefore given here.

System states are classified as in Figure 6.

Of these shown in Figure 6, a *transient state* is equivalent to the kernel running state (system call execution). From the standpoint of users, it is important that each of the system calls issued by the user application program be executed indivisibly, and that the internal states while a system call is executing cannot be seen by the user. For this reason the state while the kernel running is considered a transient state and internally it is treated as a black box.

However, in the following case, for instance, a transient state may become visible to users.

- When memory is being allocated or freed in the case of a system call that gets or releases memory (while a μ T-Kernel/SM system memory management function is called).

When a task is in a transient state such as these, the behavior of a task termination ([tk_ter_tsk](#)) system call is not guaranteed. Moreover, task suspension ([tk_sus_tsk](#)) may cause a deadlock or other problem by stopping without clearing the transient state.

Accordingly, as a rule [tk_ter_tsk](#) and [tk_sus_tsk](#) cannot be used in programs. These system calls should be used only in specific middleware or debugger, which is closely related to OS itself.

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

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

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

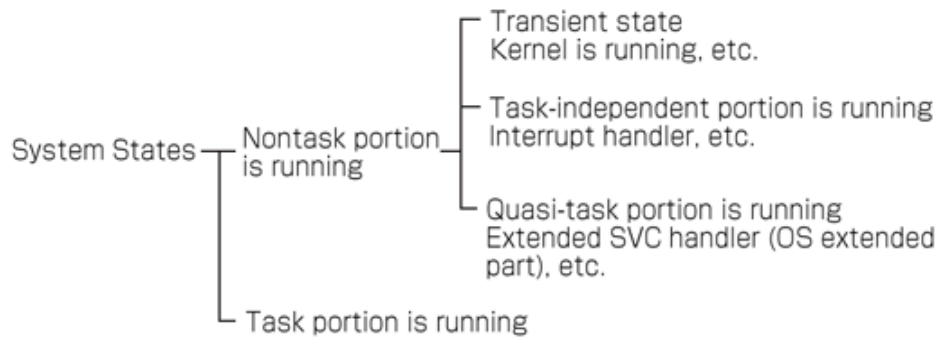


Figure 6—Classification of system states

3.5.2 Task-independent portion and quasi-task portion

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

If dispatching were to take place in the interrupt handler, which is a task-independent portion, the rest of the interrupt handler routine would be delayed for execution after the task started by the dispatching, causing problems in case of interrupt nesting. This is illustrated in Figure 7.

In Figure 7, Interrupt X is raised during Task A execution, and while its interrupt handler is running, a higher-priority interrupt Y is raised. In this case, if dispatching were to occur immediately on return from interrupt Y at (1) starting Task B, the processing of parts (2) to (3) of Interrupt X would be put off until after Task B relinquishes CPU, with parts (2) to (3) executed only after Task A goes to RUNNING state.² The danger is that the low-priority Interrupt X handler would be preempted not only by a higher-priority interrupt but even by Task B started by that interrupt. There would no longer be any guarantee of the interrupt handler execution maintaining priority over task execution, making it impossible to write an interrupt handler. This is the reason for introducing the principle of delayed dispatching.

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

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

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

- a) 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 has not yet occurred at this point. Instead, after [tk_wup_tsk](#) execution, first the remaining part of the interrupt handler is executed. Only when [tk_ret_int](#) is executed at the end of the interrupt handler does dispatching occur, causing Task B to run.
- b) An extended SVC is executed in Task A (priority 8 = low), and in its extended SVC handler (quasi-task portion), [tk_wup_tsk](#) is issued for Task B (priority 2 = high). In this case the principle of delayed dispatching is not applied, so dispatching occurs in [tk_wup_tsk](#) processing. Task A goes to READY state in a quasi-task portion, and Task B goes to RUNNING state. Task B is therefore executed before the rest of the extended SVC handler is completed. The rest of the extended SVC handler is executed after dispatching occurs again and Task A goes to RUNNING state.

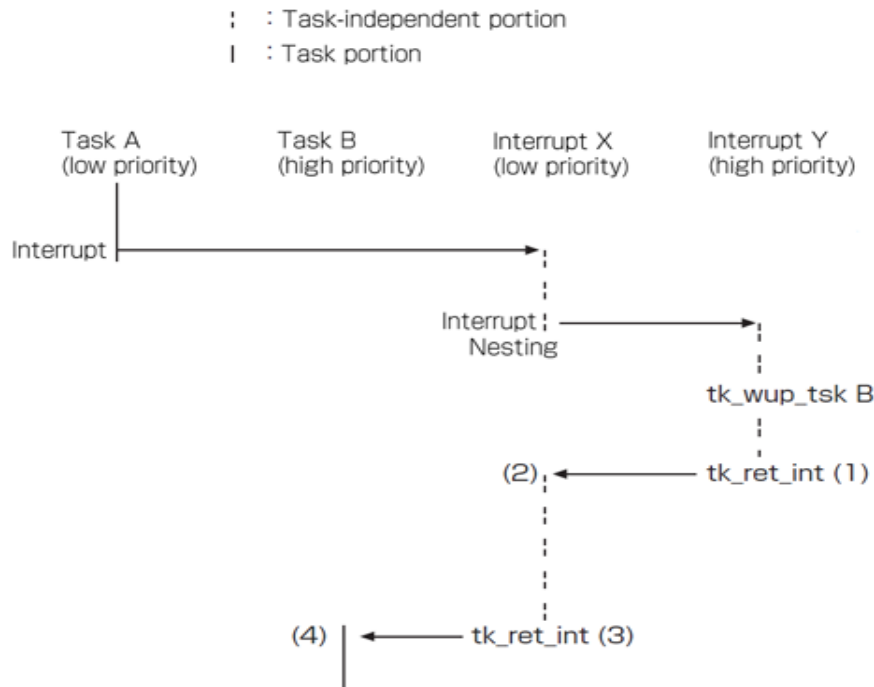


Figure 7—Interrupt nesting and delayed dispatching

3.6 Objects

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

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

In an object attribute value, the lower bits indicate system attributes and the upper bits indicate implementation-dependent attributes. This specification does not define the bit position at which the upper and lower distinction is to be made. Basically, bits that are not defined in the standard can be used as

implementation-dependent attributes. In principle, however, the system attribute portion is assigned from the least significant bit (LSB) toward the most significant bit (MSB), and implementation-dependent attributes from the MSB toward the LSB. Bits not defining any attribute must be cleared to 0.

In some cases an object may contain extended information. Extended information is specified when the object is registered. Information passed in parameters when an object starts execution has no effect on μ T-Kernel behavior. Extended information can be read by calling an object status reference system call.

An object is identified by an ID number. In μ T-Kernel, an ID number is automatically assigned when an object is created.

3.7 Protection levels

In μ T-Kernel, four levels from 0 to 3 (meaning privileged mode, user mode, etc.) are defined as the protection level at runtime, and also four levels from 0 to 3 are defined as the protection level of memory to be accessed. The currently running execution task can access only to the memory with the same or lower protection level. This function is useful for protecting a system such as the OS from being illegally accessed by programs.

The uses of each protection level are as follows:

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

Some CPUs support only two protection levels, privileged (supervisor mode) and user levels. In such a case protection level 0 is assigned to the privileged level, and protection level 3 to the user level. In such a case, if protection levels from 0 to 2 are specified in an API, the behavior of the system is the same as in the case of privileged level 0 being specified. For example, if TA_RNG2 is specified in tskatr when tk_cre_tsk is invoked, it is assumed that TA_RNG0 has been specified and the task executes at the privileged level (protection level 0). Another example is specifying TA_RNG2 in mplatr when tk_cre_mpl is invoked. This is assumed to specify TA_RNG0, and the access protection level of the created memory pool is 0. In this case, the service profile defines the following macros to be 0: TK_MEM_RNG0, TK_MEM_RNG1, TK_MEM_RNG2.

In the case of CPUs without any distinction for privileged and user modes, only protection level 0 is used. In such a case, if protection levels 1 to 3 are specified in an API, the behavior of the system is the same as in the case of privileged level 0 being specified. In this case, the service profile defines the following macros to be 0: TK_MEM_RNG0, TK_MEM_RNG1, TK_MEM_RNG2, TK_MEM_RNG3.

When a protection privilege level of the currently running context is lower than that of the memory being accessed, the violation of memory access privilege shall be detected and a CPU exception shall be generated.

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

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

3.8 Service profile

μ T-Kernel is an OS specification for small-scale embedded computer systems, and it allows many implementations and permits customization and optimization suitable for each target platform. For features that have strong dependency on hardware such as floating-point unit (FPU), and features that have potential implications for run-time efficiency such as hooks for debug support, the specification allows subsetting as an exceptional case, and this allows efficient implementation of μ T-Kernel specification OS on target hardware. To accommodate the subsetting in this manner and the desire to keep the distribution and portability of middleware and application high, μ T-Kernel has introduced a mechanism to let each implementation of μ T-Kernel describe the differences in the implementation from other implementation of μ T-Kernel. This description as a whole is called *service profile*.

Service profile in μ T-Kernel is realized by enumerating the information about a particular implementation of μ T-Kernel as a list of C language macros that have constant value. For example, an implementation that allows the specification of TA_USERBUF, the corresponding service profile items must be defined as below to announce the support of TA_USERBUF.

```
#define TK_SUPPORT_USERBUF TRUE /* Support of user-specified buffer  
                                area (TA_USERBUF) */
```

Applications and middleware can use the service profile information and write code according to the existence of the support of TA_USERBUF. For example, the following is a typical use case.

```
T_CTSK  ctsk = {  
    .exinf  = NULL,  
#if TK_SUPPORT_USERBUF  
    .tskatr = TA_HLNG|TA_RNG0|TA_USERBUF,  
    .bufptr = taskA_stack,  
#else  
    .tskatr = TA_HLNG|TA_RNG0,  
#endif  
    .task   = task,  
    .itskpri = 10,  
    .stksz  = 2048  
};  
  
tskid = tk_cre_tsk(&ctsk);
```

The preceding code sample changes, depending on the availability of TA_USERBUF, the content of parameter packet `ctsk`, which is passed to [tk_cre_tsk](#). In this manner, it is possible to develop applications and middleware that can be used on both the implementations, those that support TA_USERBUF and those that do not. Middleware developers are requested to improve the portability and thus facilitate distribution of middleware software packages by using the service profile mechanism appropriately.

For the details of service profile items defined in μ T-Kernel, see 4.4.

4. Common rules of μ T-Kernel

4.1 Data types

4.1.1 General data types

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

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

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

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

typedef INT              SZ;    /* Generic SiZe */

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

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

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

/*
 * Boolean values
 * TRUE = 1 is defined, but any value other than 0 is logically TRUE.
 * Do NOT use as in if ( bool == TRUE )
 * use as in if ( bool )
 */
```

```
*/  
typedef UINT          BOOL;  
#define TRUE          1          /* true */  
#define FALSE         0          /* false */
```

NOTE 1—VB, VH, VW, and VD differ from B, H, W, and D in that the former mean only the bit width is known, not the contents of the data type, whereas the latter clearly indicate integer type.

NOTE 2—BOOL defines TRUE as 1, but any value other than 0 is also true. For this reason, TRUE must not be used as left-hand or right-hand value of comparison operators (== and !=) for deciding whether the value is true or false. That is, conditional operations like “if (boolean value == TRUE)” should be avoided, and instead use Boolean value directly as condition, like “if (boolean value).”

NOTE 3—Parameters such as `stksz`, `wupcnt`, and message size that clearly do not take negative values are also in principle signed integer (INT or W) data type. This is in keeping with the overall TRON project rule that integers should be treated as signed numbers as much as possible. As for the timeout (TMO `tmout`) parameter, its being a signed integer enables the use of TMO_FEV (= -1) to have special meaning. Parameters with unsigned data type are those treated as bit patterns (object attribute, event flat, etc.).

4.1.1.1 Related service profile items

The 64-bit data types, D, UD, and VD, are guaranteed to be usable when the following service profile item is set to be effective.

TK_HAS_DOUBLEWORD Support of 64-bit data types (D, UD, VD)

4.1.2 Other defined data types

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

```
typedef INT           FN;          /* Function Codes */  
typedef UW           ATR;          /* Object/handler attributes */  
typedef INT           ER;          /* Error Code */  
typedef INT           PRI;         /* Priority */  
typedef W             TMO;         /* Timeout specification in milliseconds */  
typedef D             TMO_U;       /* Timeout specification in microseconds  
                                with 64-bit integer */  
  
typedef UW           RELTIM;       /* Relative time in milliseconds */  
typedef UD           RELTIM_U;     /* Relative time in microseconds  
                                with 64-bit integer */  
  
typedef struct systim {             /* System time in milliseconds */  
    W      hi;                     /* High 32 bits */  
    UW     lo;                     /* Low 32 bits */  
} SYSTIM;  
  
typedef D             SYSTIM_U;    /* System time in microseconds  
                                with 64-bit integer */  
  
/*
```

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

NOTE 1—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.

NOTE 2—The policy is to append “_u” (u means μ) or “_U” at the end for parameters and data types representing microsecond (μ s), or append “_d” (d means double integer) or “_D” at the end for other parameters and data types representing 64-bit integers. `TMO_U`, `RELTIM_U`, and `SYSTEM_U` are data type names complying to this policy.

4.1.2.1 Related service profile items

`TMO_U`, `RELTIM_U`, and `SYSTEM_U` dealing with date and relative time in microsecond resolution are guaranteed to be usable only when the following service profile items are set to be effective:

<code>TK_SUPPORT_USEC</code>	Support instructions that can handle time in microsecond resolutions
------------------------------	--

4.2 System calls

4.2.1 System call format

μ T-Kernel adopts C as the standard high-level language and standardizes interfaces for system call execution from C language routines. See Annex B for the list of C language interfaces.

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

The following common rules are established for system call interfaces:

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

The implementation of the system call interface is not standardized and is implementation-dependent. For example, we can use C language macros, inline functions, inline assembly language description.

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

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

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

- a) Include the following descriptions in the common include file:

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

- b) Describe a function definition or system call definition in the program by using CONST.

```
/* Description example of CONST */  
tk_cre_tsk( CONST T_CTSK *pk_ctsk );  
foo_bar( CONST void *buf );
```

- c) Enable `const` by the specification in **Makefile**. (Recommended)

```
# Example of enabling const  
CFLAGS += -DTKERNEL_CHECK_CONST
```

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

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

4.2.2 System calls available in task-independent portions

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

System call name	Summary description
<u>tk_sta_tsk</u>	Start task
<u>tk_wup_tsk</u>	Wakeup task
<u>tk_rel_wai</u>	Release wait
<u>tk_sus_tsk</u>	Suspend task
<u>tk_sig_sem</u>	Signal semaphore
<u>tk_set_flg</u>	Set event flag
<u>tk_sig_tev</u>	Signal task event
<u>tk_rot_rdq</u>	Rotate ready queue
<u>tk_get_tid</u>	Get task identifier
<u>tk_sta_cyc</u>	Start cyclic handler
<u>tk_stp_cyc</u>	Stop cyclic handler
<u>tk_sta_alm</u>	Start alarm handler
<u>tk_sta_alm_u</u>	Start alarm handler (in microseconds)

<u>tk_stp_alm</u>	Stop alarm handler
<u>tk_ref_tsk</u>	Reference task status
<u>tk_ref_cyc</u>	Reference cyclic handler status
<u>tk_ref_cyc_u</u>	Reference cyclic handler status (microseconds)
<u>tk_ref_alm</u>	Reference alarm handler status
<u>tk_ref_alm_u</u>	Reference alarm handler status (microseconds)
<u>tk_ref_sys</u>	Reference system status
<u>tk_ret_int</u>	Return from interrupt handler (can be issued only from an interrupt handler written in an assembly language)

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

4.2.3 Restricting system call invocation

The protection levels at which a system call is invokable can be restricted. In this case, if a system call is issued from a task (task portion) running at lower privilege than the specified protection level, the error code E_OACV is returned.

Extended SVC calling cannot be restricted.

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

This kind of restriction is used when μ T-Kernel is combined with middleware that offers process management function and other functions, to prevent tasks (as part of user process, etc.) that use the functions of such middleware (process management, etc.) from directly accessing μ T-Kernel functions. It allows μ T-Kernel to be used as a micro-kernel. The idea is that the user process cannot control the micro-kernel directly via available process API, and only the middleware can control the micro-kernel directly.

The protection level restriction on system call invocation is set using the system configuration information management functions. (see 6.6.)

4.2.4 Modifying a parameter packet format

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

Additional information that is implementation-dependent can be added to a parameter packet. When implementation-dependent information is added, it must be positioned after the standard defined information. It is permitted to delete only parameters that are declared ineffective by the service profile, and other parameters shall not be deleted. It is not allowed, however, to change the data types and order of information defined in this specification.

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

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

The reason for this specification is to make sure we can run the same application program merely by recompiling, irrespective of whether implementation dependent function extension is added to an implementation of the specification..

4.2.4.1 Porting guideline

Care must be taken for parameter packet initialization since the parameter may be deleted by declaring it to be ineffective by service profile. For example, it is not recommended to initialize T_CTSK structure in the following manner from the viewpoint of portability.

```
T_CTSK  ctsk = {  
    NULL,  
    TA_HLNG|TA_RNG0|TA_USERBUF,  
    task,  
    10,  
    2048,  
    "",  
    buf  
};
```

Instead, it is recommended to perform initialization using the syntax specified in ISO-IEC 9899:1999 as follows:

```
T_CTSK  ctsk = {  
    .exinf    = NULL,  
    .tskatr   = TA_HLNG|TA_RNG0|TA_USERBUF,  
    .task     = task,  
    .itskpri  = 10,  
    .stksz    = 2048,  
    .bufptr   = buf  
};
```

4.2.5 Function codes

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

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

4.2.6 Error codes

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

An error code consists of the main error code and sub error code. The low 16 bits of the error code are the sub error code, and the remaining high bits are the main error code. Main error codes are classified into error classes based on the necessity of their detection, the circumstances in which they occur and other factors.

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

Note that, in an environment where ER is 16-bit data type, sub error code can be omitted and main error code can be returned as the error code. In this case, SERCD macro shall not be defined.

```
#define MERCD(er)      ( (ER)(er) )              /* main error code */
#define ERCD(mer, ser) ( (ER)(mer) )
```

4.2.6.1 Related service profile items

Only when the following service profile items are set to be effective, the error code contains sub error code, and SERCD macro is supported.

TK_SUPPORT_SERCD	Support of sub error code
------------------	---------------------------

4.2.7 Timeout

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

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

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

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

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

NOTE 1—Since a system call that performs polling does not enter WAITING state, there is no change in the precedence of the task calling it.

NOTE 2—In a general implementation, when the timeout is set to 1, timeout processing takes place on the second timer interrupt (sometimes called “time tick”) after a system call is invoked. Since a timeout of 0 cannot be specified (0 being allocated to TMO_POL), in this kind of implementation timeout does not occur on the initial timer interrupt after the system call is invoked.

4.2.8 Relative time and system time

When the time of an event occurrence is specified relative to another time, such as the time when a system call was invoked, relative time (RELTIM or RELTIM_U type) is used. If relative time is used to specify event occurrence time, it is necessary to guarantee that the event processing will take place after the specified time has elapsed from the time base. Relative time (RELTIM or RELTIM_U type) is also used for event occurrence. In such cases the method of interpreting the specified relative time is determined for each case. When time is specified as an absolute value, system time (SYSTIM or SYSTIM_U type) is used. The μ T-Kernel provides a function for setting system time, but even if the system time is changed using this function, there is no change in the real world time (actual time) at which an event occurs that was specified using relative time. What changes is the system time at which an event occurs that was specified as relative time.

SYSTIM: System time

Time base 1 ms, 64-bit signed integer

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

SYSTIM_U: System time

Time base 1 μ s, 64-bit signed integer

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

RELTIM: Relative time

Time base 1 ms, 32-bit unsigned integer (UW)

```
typedef UW     RELTIM;
```

RELTIM_U: Relative time

Time base 1 μ s, 64-bit unsigned (UD) integer

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

TMO: Timeout time

Time base 1 ms, 32-bit signed integer (W)

```
typedef W      TMO;
```

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

TMO_U timeout period

Time base 1 μ s, 64-bit signed (D) integer

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

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

4.2.8.1 Related service profile items

TMO_U, RELTIM_U, and SYSTEM_U dealing with date and relative time in microsecond resolution are guaranteed to be usable only when the following service profile items are set to be effective.

TK_SUPPORT_USEC Support instructions that can handle time in microsecond resolutions

NOTE 1—Timeout or other such processing must be guaranteed to occur after the time specified as RELTIM, RELTIM_U, TMO, or TMO_U has elapsed. For example, if the timer interrupt interval is 1 ms and a timeout of 1 ms is specified, timeout occurs on the second timer interrupt after system call invocation (the first timer interrupt does not exceed 1 ms).

NOTE 2—When a system time (SYSTIM_U) value that may overflow internally in kernel is specified as an argument, the system call behavior is undefined.

4.3 High-level language support routines

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

When TA_HLNG is not specified, a task or handler is started directly from the start address passed in a parameter to [tk_cre_tsk](#), [tk_def_int](#), etc.; 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](#), [tk_def_int](#), etc. Viewed from the kernel, the task start address or handler address is a parameter given to the high-level language support routine. Separating the kernel processing from the language environment processing in this way facilitates support for different language environments.

Use of high-level language support routines has the further advantage that when a handler is written as a C language function, return from a handler can be executed automatically, simply by performing a function return.

In a system that utilizes CPU's operating modes, however, whereas it is relatively easy to realize a high-level language support routine in the case of an interrupt handler or the like that runs at the same protection level as the kernel, it is more difficult in the case of a task or task exception handler running at a different protection level from the kernel's. For this reason, when a high-level language support routine is used for a task, there is no guarantee that the task will exit by a return from the function. Returning a task function using `return` or `}"` leads to an undefined behavior. At the end of a task, Exit task ([tk_ext_tsk](#)) or Exit and delete task ([tk_exd_tsk](#)) must always be issued.

In the case of a task exception handler, the high-level language support routine is supplied as source code and is to be embedded in the user program.

The internal working of a high-level language support routine is as illustrated in Figure 8.

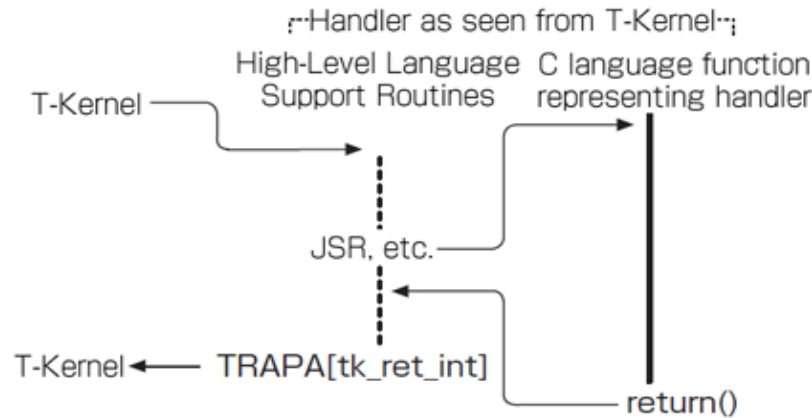


Figure 8—Behavior of high-level language support routine

4.4 Service profile

See Annex D for the list of APIs and service profile names.

4.4.1 Service profile items that represent function availability

The service profile item that shows whether a particular function is effective (or enabled) or ineffective (or disabled) is described by using the following macro, which is defined to be either TRUE or FALSE. (The following definitions are given as examples only, and each implementation shall define these appropriately.)

4.4.1.1 Device driver functions

```
#define TK_SUPPORT_TASKEVENT TRUE /* Support of task event */
#define TK_SUPPORT_DISWAI TRUE /* Support of disabling wait */
#define TK_SUPPORT_IOPORT TRUE /* Support of I/O port access */
#define TK_SUPPORT_MICROWAIT TRUE /* Support of micro wait */
```

4.4.1.2 Power management functions

```
#define TK_SUPPORT_LOWPOWER TRUE /* Support of power management */
```

4.4.1.3 Static/dynamic memory management functions

```
#define TK_SUPPORT_USERBUF TRUE /* Support of user-specified buffer
                                (TA_USERBUF) */
#define TK_SUPPORT_AUTOBUF TRUE /* Support of automatic buffer allocation
                                (No TA_USERBUF specification) */
#define TK_SUPPORT_MEMLIB TRUE /* Support of memory allocation library */
```

You cannot set both TK_SUPPORT_USERBUF and TK_SUPPORT_AUTOBUF to FALSE.

4.4.1.4 Task exception handling functions

```
#define TK_SUPPORT_TASKEXCEPTION TRUE /* Support of task exception */
```

4.4.1.5 System configuration information acquisition functions

```
#define TK_SUPPORT_SYSCONF TRUE /* Support of getting system  
configuration */
```

4.4.1.6 Supporting 64-bit and 16-bit CPUs

```
#define TK_HAS_DOUBLEWORD TRUE /* Support of 64-bit data types  
(D, UD, VD) */  
#define TK_SUPPORT_USEC TRUE /* Support of microsecond */  
#define TK_SUPPORT_LARGEDEV TRUE /* Support of Large Mass-storage Device  
(64-bit) */  
#define TK_SUPPORT_SERCD TRUE /* Support of sub error code */
```

TK_HAS_DOUBLEWORD, TK_SUPPORT_USEC, and TK_SUPPORT_LARGEDEV need to be set to either TRUE or FALSE according to the target hardware characteristics and the usage purpose of the target system.

NOTE—TK_SUPPORT_USEC and TK_SUPPORT_LARGEDEV depend on TK_HAS_DOUBLEWORD. That is, when TK_HAS_DOUBLEWORD is set to be ineffective, these two profile items are also set to be ineffective.

4.4.1.7 Functions that depend on CPU, hardware, system, and compiler

Each of the following profiles needs to be set to TRUE or FALSE according to the target hardware and the implementation of the OS.

4.4.1.7.1 Interrupt management functions

```
#define TK_SUPPORT_INTCTRL TRUE /* Support of interrupt controller  
management */  
#define TK_HAS_ENAINTLEVEL TRUE /* Can specify interrupt priority  
level */  
#define TK_SUPPORT_CPUINTLEVEL TRUE /* Support of get/set of CPU interrupt  
mask level */  
#define TK_SUPPORT_CTRLINTLEVEL TRUE /* Support of get/set of interrupt  
controller interrupt mask level */  
#define TK_SUPPORT_INTMODE TRUE /* Support of setting interrupt mode */
```

4.4.1.7.2 Memory cache control functions

```
#define TK_SUPPORT_CACHECTRL TRUE /* Support of cache control */
#define TK_SUPPORT_WBCACHE TRUE /* Support of write-back cache */
#define TK_SUPPORT_WTCACHE TRUE /* Support of write-through cache */
```

4.4.1.7.3 FPU (COP) support functions

```
#define TK_SUPPORT_FPU TRUE /* Support of FPU */
#define TK_SUPPORT_COP0 TRUE /* Support of co-processor number 0 */
#define TK_SUPPORT_COP1 TRUE /* Support of co-processor number 1 */
#define TK_SUPPORT_COP2 TRUE /* Support of co-processor number 2 */
#define TK_SUPPORT_COP3 TRUE /* Support of co-processor number 3 */
```

4.4.1.7.4 Miscellaneous functions

```
#define TK_SUPPORT_ASM TRUE /* Support of assembly language function
                             entry/exit */
#define TK_SUPPORT_REGOPS TRUE /* Support of get/set register
                                operation */
#define TK_ALLOW_MISALIGN TRUE /* Memory misalign access is permitted */
#define TK_BIGENDIAN TRUE /* Is Big Endian (Must be defined) */
#define TK_TRAP_SVC TRUE /* Use CPU Trap instruction for system
                           call entry */
#define TK_HAS_SYSSTACK TRUE /* Task has a separate system stack */
#define TK_SUPPORT_PTIMER TRUE /* Support of physical timer */
```

4.4.1.8 Check method of service profile

Although the implementations of μ T-Kernel must define the profile items mentioned previously, the use of profile where some definitions are missing should be practiced since other OSs does not provide profile at all, and there bound to be implementation's failures to define all the profile items. For example, if you want to distinguish the effective/ineffective/undefined status, you can perform the following check:

```
#if defined(TK_SUPPORT_xxx)
    #if TK_SUPPORT_xxx
        /* when a profile item is set to be effective. */
    #else
        /* when a profile item is set to be ineffective */
    #endif
#else
    /* when a profile item is undefined. */
#endif
```

Note that if profile item is directly used for the parameter of “#if” macro as follows, you cannot distinguish whether the profile item is ineffective or undefined.

```
#if TK_SUPPORT_xxx
    /* when a profile item is set to be effective. */
#else
```

```
/* when a profile item is set to be ineffective or undefined. */
#endif
```

4.4.2 Service profile items that represent values

A service profile item that represents a limit value or version number will be specified as a MACRO that holds the value. (The following definitions are given as example only. The real values of profile items are implementation-dependent.)

```
#define TK_SPECVER_MAGIC      6      /* Magic Number of  $\mu$ T-Kernel */
#define TK_SPECVER_MAJOR      2      /* Major Version number of the
                                       Specification */
#define TK_SPECVER_MINOR      0      /* Minor version number of the
                                       Specification */
#define TK_SPECVER            ((TK_SPECVER_MAJOR << 8) | TK_SPECVER_MINOR)
                               /* Version number of  $\mu$ T-Kernel */

#define TK_MAX_TSKPRI         32      /* Maximum task priority ( $\geq 16$ ) */
#define TK_WAKEUP_MAXCNT      65535  /* Maximum queuing number of the task
                                       wakeup requests ( $\geq 1$ ) */
#define TK_SEMAPHORE_MAXCNT   65535  /* Maximum semaphore count ( $\geq 32767$ ) */
#define TK_SUSPEND_MAXCNT     65535  /* Maximum queuing count of the
                                       forced wait of tasks */
#define TK_MEM_RNG0           0      /* Real memory protection level of
                                       TA_RNG0 (0~3) */
#define TK_MEM_RNG1           0      /* Real memory protection level of
                                       TA_RNG1 (0~3) */
#define TK_MEM_RNG2           0      /* Real memory protection level of
                                       TA_RNG2 (0~3) */
#define TK_MEM_RNG3           3      /* Real memory protection level of
                                       TA_RNG3 (0~3) */
#define TK_MAX_PTIMER         2      /* Maximum number of physical timers
                                       ( $\geq 0$ ) (Values from 1 to
                                       TK_MAX_PTIMER can be used as
                                       physical timer number) */
```

TK_MEM_RNG n defines the real memory protection level of memory specified by TA_RNG n , and if TK_MEM_RNG n == TK_MEM_RNG m , then as far as memory access protection level goes, TA_RNG n and TA_RNG m are equivalent. In other words, it is guaranteed that a task with protection level m can access memory with protection level n without generating access privilege violation exception.

It is recommended that the developer is prepared for the case of missing definitions for service profile items that are supposed to have a value by means of coding such as defined(...).

4.4.3 Examples of service profile items

Following are concrete examples of service profile items.

4.4.3.1 Service profile items for a very small-scale system using 16-bit CPU

```
#define TK_SUPPORT_TASKEVENT      FALSE
#define TK_SUPPORT_DISWAI        FALSE
#define TK_SUPPORT_IOPORT        TRUE
#define TK_SUPPORT_MICROWAIT      TRUE

#define TK_SUPPORT_LOWPOWER      TRUE

#define TK_SUPPORT_USERBUF        TRUE
#define TK_SUPPORT_AUTOBUF        FALSE
#define TK_SUPPORT_MEMLIB        FALSE

#define TK_SUPPORT_TASKEXCEPTION  FALSE

#define TK_SUPPORT_SYSCONF        FALSE

#define TK_HAS_DOUBLEWORD        FALSE
#define TK_SUPPORT_USEC          FALSE
#define TK_SUPPORT_LARGEDEV       FALSE
#define TK_SUPPORT_SERCD         FALSE

#define TK_SUPPORT_INTCTRL        FALSE
#define TK_HAS_ENAINTLEVEL        FALSE
#define TK_SUPPORT_CPUINTLEVEL    FALSE
#define TK_SUPPORT_CTRLINTLEVEL   FALSE
#define TK_SUPPORT_INTMODE        TRUE

#define TK_SUPPORT_CACHECTRL      FALSE
#define TK_SUPPORT_WBCACHE        FALSE
#define TK_SUPPORT_WTCACHE        FALSE

#define TK_SUPPORT_FPU            FALSE
#define TK_SUPPORT_COP0           FALSE
#define TK_SUPPORT_COP1           FALSE
#define TK_SUPPORT_COP2           FALSE
#define TK_SUPPORT_COP3           FALSE

#define TK_SUPPORT_ASM            TRUE
#define TK_SUPPORT_REGOPS         FALSE
#define TK_ALLOW_MISALIGN         FALSE
#define TK_BIGENDIAN              FALSE
#define TK_TRAP_SVC               FALSE
#define TK_HAS_SYSSTACK           FALSE
#define TK_SUPPORT_PTIMER         FALSE

#define TK_SPECVER_MAJOR          2
#define TK_SPECVER_MINOR          0
#define TK_SPECVER                ((TK_SPECVER_MAJOR << 8) | TK_SPECVER_MINOR)
#define TK_MAX_TSKPRI             16
#define TK_WAKEUP_MAXCNT          4095
#define TK_SEMAPHORE_MAXCNT       4095
#define TK_SUSPEND_MAXCNT         4095
#define TK_MEM_RNG0               0
```

```
#define TK_MEM_RNG1          0
#define TK_MEM_RNG2          0
#define TK_MEM_RNG3          0
#define TK_MAX_PTIMER        0
```

4.4.3.2 Service profile items for a relatively large-scale system

```
#define TK_SUPPORT_TASKEVENT    TRUE
#define TK_SUPPORT_DISWAI      TRUE
#define TK_SUPPORT_IOPORT      TRUE
#define TK_SUPPORT_MICROWAIT    TRUE

#define TK_SUPPORT_LOWPPOWER    TRUE

#define TK_SUPPORT_USERBUF      FALSE
#define TK_SUPPORT_AUTOBUF      TRUE
#define TK_SUPPORT_MEMLIB       TRUE

#define TK_SUPPORT_TASKEXCEPTION TRUE
#define TK_SUPPORT_SYSCONF      TRUE

#define TK_HAS_DOUBLEWORD      TRUE
#define TK_SUPPORT_USEC         TRUE
#define TK_SUPPORT_LARGEDEV     TRUE
#define TK_SUPPORT_SERCD        TRUE

#define TK_SUPPORT_INTCTRL      TRUE
#define TK_HAS_ENAINTLEVEL      TRUE
#define TK_SUPPORT_CPUINTLEVEL  FALSE
#define TK_SUPPORT_CTRLINTLEVEL TRUE
#define TK_SUPPORT_INTMODE      TRUE

#define TK_SUPPORT_CACHECTRL    TRUE
#define TK_SUPPORT_WBCACHE      TRUE
#define TK_SUPPORT_WTCACHE      TRUE

#define TK_SUPPORT_FPU          TRUE
#define TK_SUPPORT_COP0         TRUE
#define TK_SUPPORT_COP1         FALSE
#define TK_SUPPORT_COP2         FALSE
#define TK_SUPPORT_COP3         FALSE

#define TK_SUPPORT_ASM          TRUE
#define TK_SUPPORT_REGOPS       TRUE
#define TK_ALLOW_MISALIGN       FALSE
#define TK_BIGENDIAN            FALSE
#define TK_TRAP_SVC             TRUE
#define TK_HAS_SYSSTACK         TRUE
#define TK_SUPPORT_PTIMER       TRUE

#define TK_SPECVER_MAJOR        2
#define TK_SPECVER_MINOR        0
```

```
#define TK_SPECVER ((TK_SPECVER_MAJOR << 8) | TK_SPECVER_MINOR)
#define TK_MAX_TSKPRI 140
#define TK_WAKEUP_MAXCNT 65535
#define TK_SEMAPHORE_MAXCNT 65535
#define TK_SUSPEND_MAXCNT 65535
#define TK_MEM_RNG0 0
#define TK_MEM_RNG1 0
#define TK_MEM_RNG2 0
#define TK_MEM_RNG3 3
#define TK_MAX_PTIMER 10
```

4.5 API notation

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

4.5.1 API name—Description

This is an API name and its short description.

4.5.2 C Language interface

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

4.5.3 Parameter

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

4.5.4 Return parameter

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

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

4.5.5 Error codes

Describes errors that can occur in an API.

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

E_SYS, E_NOSPT, E_RSFN, E_MACV, E_OACV.

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

E_PAR, E_MACV, E_CTX.

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

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

4.5.6 Valid context

Indicates the context (task portion, quasi-task portion, and task-independent portion) that can issue the API under consideration. Note that items marked with “x” are sometimes clearly impossible to use in the context discussed, but the usability of some items in the context discussed may be implementation-dependent, and some may be usable in some implementations.

4.5.7 Related service profile items

The relation of the service profile item(s) associated with API is shown.

4.5.8 Description

Describes the API functions.

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

$(x \parallel y \parallel z)$

Set one of x, y, or z.

$x \mid y$

Both x and y can be set at the same time (in which case the logical sum of x and y is taken).

$[x]$

x is optional.

Example of using parameters notation

$\text{wfmode} := (\text{TWF_ANDW} \parallel \text{TWF_ORW}) \mid [\text{TWF_CLR}]$

The preceding description means that wfmode can be specified in any of the following four ways:

TWF_ANDW

TWF_ORW

(TWF_ANDW | TWF_CLR)

(TWF_ORW | TWF_CLR)

4.5.9 Additional notes

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

4.5.10 Rationale for the specification

Explains the reason for adopting a particular approach and specification.

5. μ T-Kernel/OS functions

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

5.1 Task management functions

Task management functions are functions that directly manipulate or reference task states. Functions are provided for creating and deleting a task, for task starting and exit, changing task priority, and referencing task state. A task is an object identified by an ID number called a *task ID*. Task states and scheduling rules are explained in 3.2.

For control of execution order, a task has a base priority and current priority. When simply “task priority” is mentioned, this means the current priority. The base priority of a task is initialized to the startup priority when a task is started. If the mutex function is not used, the task current priority is always identical to its base priority. For this reason, the current priority immediately after a task is started is the task startup priority. When the mutex function is used, the current priority is set as discussed in 5.5.1.

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

5.1.1 tk_cre_tsk—Create task

5.1.1.1 C Language interface

```
#include <tk/kernel.h>
ID tskid = tk_cre_tsk( CONST T_CTSK *pk_ctsk );
```

5.1.1.2 Parameter

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

pk_ctsk detail:

void*	exinf	Extended Information	Extended information
ATR	tskatr	Task Attribute	Task attribute
FP	task	Task Start Address	Task start address
PRI	itskpri	Initial Task Priority	Initial task priority
SZ	stksz	Stack Size	Stack size (in bytes)
SZ	sstksz	System Stack Size	System stack size (in bytes)
void*	stkptr	User Stack Pointer	User stack pointer
void*	bufptr	Buffer Pointer	User buffer pointer

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

5.1.1.3 Return parameter

ID	tskid	Task ID	Task ID
		or Error Code	Error code

5.1.1.4 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 coprocessor does not exist
E_NOSPT	Unsupported functions(when the specification of TA_ASM, TA_USERSTACK, TA_TASKSPACE, or TA_USERBUF is not supported.)
E_PAR	Parameter error
E_NOCOP	The specified coprocessor cannot be used (not installed, or abnormal operation detected)

5.1.1.5 Valid context

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

5.1.1.6 Related service profile items

TK_SUPPORT_ASM	Support for specifying TA_ASM for task attribute
TK_SUPPORT_USERBUF	Support for specifying TA_USERBUF for task attribute

TK_SUPPORT_AUTOBUF	Automatic buffer allocation is supported (by not specifying TA_USERBUF to task attribute)
TK_SUPPORT_FPU	Support for specifying TA_FPU for task attribute
TK_SUPPORT_COPn	Support for specifying TA_COPn for task attribute
TK_HAS_SYSSTACK	Task can have a system stack independent of user-stack, and each can be specified separately using (TA_USERSTACK, TA_SSTKSZ)
TK_MAX_TSKPRI	Maximum task priority that can be specified (must be 16 or higher)

5.1.1.7 Description

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

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

`itskpri` is used to specify the startup priority when a task is started. Task priority level can be specified by a positive integer, and the smaller the value, the higher priority the task has. The largest task priority level is defined by `TK_MAX_TSKPRI`.

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

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

```
tskatr := (TA_ASM || TA_HLNG) | [TA_SSTKSZ] | [TA_USERSTACK] | [TA_USERBUF]
          | (TA_RNG0 || TA_RNG1 || TA_RNG2 || TA_RNG3)
          | [TA_COP0] | [TA_COP1] | [TA_COP2] | [TA_COP3] | [TA_FPU]
```

TA_ASM	Indicates that the task is written in assembly language
TA_HLNG	Indicates that the task is written in high-level language
TA_SSTKSZ	Specifies the system stack size
TA_USERSTACK	Points to the user stack
TA_USERBUF	Use of user-specified memory area as stack
TA_RNGn	Indicates that the task runs at protection level n
TA_COPn	Specifies use of the nth coprocessor (including floating-point coprocessor or DSP)
TA_FPU	Specifies use of a floating-point coprocessor (when a coprocessor specified in <code>TA_COPn</code> is a general-purpose FPU particularly for floating-point processing and not dependent on the CPU)

The function for specifying implementation-dependent attributes can be used, for example, to specify that a task is subject to debugging. One use of the remaining system attribute fields is for indicating multiprocessor attributes in the future.

```
#define TA_ASM          0x00000000    /* Task in Assembly Language */
#define TA_HLNG         0x00000001    /* Task in High-level language */
#define TA_SSTKSZ       0x00000002    /* System stack size */
#define TA_USERSTACK    0x00000004    /* User stack pointer */
#define TA_USERBUF      0x00000020    /* Use user-specified buffer */
#define TA_RNG0         0x00000000    /* Run at protection level 0 */
#define TA_RNG1         0x00000100    /* Run at protection level 1 */
#define TA_RNG2         0x00000200    /* Run at protection level 2 */
#define TA_RNG3         0x00000300    /* Run at protection level 3 */
#define TA_COP0         0x00001000    /* Use ID=0 coprocessor */
#define TA_COP1         0x00002000    /* Use ID=1 coprocessor */
#define TA_COP2         0x00004000    /* Use ID=2 coprocessor */
#define TA_COP3         0x00008000    /* Use ID=3 coprocessor */
```

When `TA_HLNG` is specified, starting the task jumps to the task address not directly but by going through a high-level language environment configuration program (high-level language support routine). The task takes the following form in this case:

```
void task( INT stacd, void *exinf )
{
    /*
        (processing)
    */

    tk_ext_tsk(); or tk_extd_tsk(); /* Exit task */
}
```

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

The task cannot (must not) be terminated by a simple return from the function, otherwise the operation will be indeterminate (implementation-dependent).

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

The task runs at the protection level specified in the `TA_RNGn` attribute. When a system call or extended SVC is called, the protection level goes to 0, then goes back to its original level upon return from the system call or extended SVC.

Each task has two stack areas, a system stack and user stack. The user stack is used at the protection level specified in `TA_RNGn` while the system stack is used at protection level 0. When the calling of a system call or extended SVC causes the protection level to change, the stack is also switched.

Note that a task running at `TA_RNG0` does not switch protection levels, so there is no stack switching either. When `TA_RNG0` is specified, the combined total of the user stack size and system stack size is the size of one stack, employed as both a user stack and system stack.

When `TA_SSTKSZ` is specified, `sstksz` is valid. If `TA_SSTKSZ` is not specified, `sstksz` is ignored and the default size applies.

When `TA_USERSTACK` is specified, `stkptr` is valid. In this case a user stack is not provided by the OS, but must be allocated by the caller. `stksz` must be set to 0. If `TA_USERSTACK` is not specified, `stkptr` is ignored. Note that if `TA_RNG0` is set, `TA_USERSTACK` cannot be specified. `E_PAR` occurs if `TA_RNG0` and `TA_USERSTACK` are specified at the same time.

`TA_USERBUF` can be specified for implementation where there is no distinction of user stack and system stack and there is only one unified stack for a task. When this attribute is specified, `bufptr` becomes effective, and the memory area starting at `bufptr` containing `stksz` octets is used as the unified user and system stack area. In this case, the kernel does not provide the stack area.

5.1.1.8 Additional notes

A task runs either at the protection level set in `TA_RNGn` or at protection level 0. For example, a task for which `TA_RNG3` is specified in no case runs at protection level 1 or 2.

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

The system stack default size is decided, taking into account the amount taken up by system call execution and, in a system with separate interrupt stack, the amount used by interrupt handlers.

The definition of `TA_COPn` is dependent on the CPU and other hardware and is not portable.

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

Even in a system with a single CPU's operating mode, for the sake of portability all attributes including `TA_RNGn` must be accepted. It is possible, for example, to handle all `TA_RNGn` as equivalent to `TA_RNG0`, but error must not be returned.

5.1.2 `tk_del_tsk`—Delete task

5.1.2.1 C Language interface

```
#include <tk/tkernel.h>
ER ercd = tk_del_tsk( ID tskid );
```

5.1.2.2 Parameter

ID	<code>tskid</code>	Task ID	Task ID
----	--------------------	---------	---------

5.1.2.3 Return parameter

ER	<code>ercd</code>	Error Code	Error code
----	-------------------	------------	------------

5.1.2.4 Error codes

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

5.1.2.5 Valid context

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

5.1.2.6 Related service profile items

None.

5.1.2.7 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.

5.1.3 tk_sta_tsk—Start task

5.1.3.1 C Language interface

```
#include <tk/tkernel.h>
ER ercd = tk_sta_tsk( ID tskid, INT stacd );
```

5.1.3.2 Parameter

ID	<code>tskid</code>	Task ID	Task ID
INT	<code>stacd</code>	Task Start Code	Task start code

5.1.3.3 Return parameter

ER	<code>ercd</code>	Error Code	Error code
----	-------------------	------------	------------

5.1.3.4 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)

5.1.3.5 Valid context

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

5.1.3.6 Related service profile items

None.

5.1.3.7 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.

5.1.3.8 Porting guideline

Note that **stacd** is INT type, and its value range is implementation-dependent, so care must be taken.

5.1.4 tk_ext_tsk—Exit task

5.1.4.1 C Language interface

```
#include <tk/kernel.h>
void tk_ext_tsk(void);
```

5.1.4.2 Parameter

None.