μITRON4.0 Specification

Ver. 4.00.00

ITRON Committee, TRON ASSOCIATION

Supervised by Ken Sakamura

Edited by Hiroaki Takada

µITRON4.0 Specification (Ver. 4.00.00)

Any questions regarding this specification and this specification document should be directed to the following:

ITRON Committee, TRON Association
Katsuta Building 5F
3-39, Mita 1-chome, Minato-ku,
Tokyo 108-0073, JAPAN
TEL: +81-3-3454-3191
FAX: +81-3-3454-3224

---

§ TRON is the abbreviation of "The Real-time Operating system Nucleus."
§ ITRON is the abbreviation of "Industrial TRON."
§ µITRON is the abbreviation of "Micro Industrial TRON."
§ BTRON is the abbreviation of "Business TRON."
§ CTRON is the abbreviation of "Central and Communication TRON."
§ TRON, ITRON, µITRON, BTRON, and CTRON do not refer to any specific product or products.

A Word from the Project Leader

Fifteen years have passed since the ITRON Sub-Project started as a part of the TRON Project: a real-time operating system specification for embedded equipment control. During this time, there has been a high degree of technological innovation on microprocessors, and the range of applications the ITRON Specifications cover has broadened considerably. The range of applications includes industrial usage such as control of robots and manufacturing equipment in factories, and consumer usage such as office automation (OA) and home appliances. The application range has even extended to new areas such as new information and communication tools and advanced digital consumer appliances. There is no doubt that the technological advantages of the ITRON Specifications such as real-time response, compactness to maximize usage of system resources, and flexible adaptability in specification has greatly contributed to the steady expansion of the ITRON Specifications adaptable applications. The open architecture policy of the TRON Project has also contributed to achieve a high degree of actual use of the ITRON Specifications.

The μITRON4.0 Specification, which is based on the μITRON3.0 Specification, has been developed to reorganize concepts and terms, to improve compatibility and conformance level, to increase productivity in software development, to allow reuse of application software, and to achieve more portability.

The increasing cases where communication and GUI focusing on modern network applications, internet and intranet equipments, and debugging related middleware are used on the ITRON-Specification operating system serves as the background of the μITRON4.0 Specification. This trend created a demand for more rigorous compatibility and higher conformance level. The ITRON Specifications are designed on a concept called loose standardization and level for allowing applicability to low-end CPUs with relatively scarce resources. However, strict standardization is required for software portability purposes. The specification satisfying these two contradictory demands is the μITRON4.0 Specification. The μITRON4.0 Specification maintains the loose standardization, but develops the level concept to introduce a new property called the Standard Profile. The Standard Profile supports strict standardization to facilitate software portability. Profiles other than the Standard Profile is allowed to increase compatibility in each application field.

The terms and concepts in the specification have been reorganized, defined, and explained in more details, reducing as much implementation-dependent portion as possible in an effort to achieve completeness of the specification.

The μITRON4.0 Specification reflects the rich experience on the ITRON Specifications and meets the actual users' demands for new applications. Introduction and effective utilization of the μITRON4.0 Specification in many fields including newly created fields and applications is expected.

June, 1999

Ken Sakamura
Project Leader, TRON Project

Preface

Fifteen years have elapsed since the ITRON Project started in 1984. By the efforts of those involved, the μITRON Specifications have developed into de-facto standards for the real-time kernel for embedded systems. Based on this achievement, sometime around 1996 the ITRON Project started working towards a second phase of standardization to expand the specification from real-time kernel to related specifications such as software components.

The μITRON4.0 Specification is the result of two years of intensive effort by both the Kernel Specification WG of the Hard Real-Time Support Study Group (from April 1997 to March 1998) and its successor, the Kernel Specification WG of the μITRON4.0 Specification Study Group (from April 1998 to June 1999). The specification study held by both WGs was proceeded with monthly meetings and discussions through email. Eventually, more than 1,000 emails regarding the discussion of the specification were exchanged. Other standardization activities in the ITRON Project, especially those done by the RTOS Automotive Application Technical Committee and the Device Driver Design Guideline WG of the μITRON4.0 Specification Study Group, produced an important part of the μITRON4.0 Specification.

During the second phase of the standardization, a new approach was adopted by the ITRON Project. The ITRON Project opened the discussion of the μITRON4.0 Specification. In other words, anyone could participate in the discussion regardless of qualification. This approach was a major factor in enabling many engineers to participate in the project. Prior to this, most of the engineers did not formerly participate in the discussion. The participation of application engineers as well as the participation of kernel engineers was very significant in organizing the specification.

Another new attempt of the μITRON4.0 Specification is defining the Standard Profile in order to insure the portability of the software. Under conventional loose standardization policy, there is no enforced implementation agreement among members. Compromises are adopted, and it is up to the implementors to choose specific implementation options. However, when defining the Standard Profile specification, standardizing each and every feature of the μITRON Specifications was necessary and this caused many disagreements among the members. Most disagreements were based on the difference between application requirements rather than on the difference between company interests. Nevertheless, members shared a common vision to create a better specification.

Through the process outlined above, the μITRON4.0 Specification was completed reflecting variety of ideas from a variety of point of views. I am personally proud of the level of accomplishment of the μITRON4.0 Specification. I believe that this level of accomplishment could not have been achieved by a single man or company.

To the readers of the μITRON4.0 Specification, I would like to remark that in the interests of adhering to strictness in the specification, some readability was sacrificed. The previous µITRON Specifications included tutorial-like contents for engineers who are unfamiliar with a real-time operating system. On the other hand, the µITRON4.0 Specification is written seeking strictness rather than easiness in reading in order to secure software portability. Hence, a criticism on a lesser understandability than the previous specifications is considered to be unavoidable. Therefore we would like to work on some complementary documents such as a reference or a guide book for this specification. However, the editor's responsibility still spans to statements which are unnecessarily difficult to understand.

As a roadmap for the ITRON Specifications, the µITRON4.0 Specification Study Group is working on the standardization of the debugging interface and creating guidelines for device driver designs. The creation of a certification system for the µITRON4.0 Standard Profile is also under consideration in the near future. We are sure that these activities will increase the acceptance of the ITRON Specifications as de-facto real-time operating system standards.

Finally, I would like to express my gratitude to those who contributed to the standardization of the µITRON4.0 Specification. This includes those who participated in the Kernel Specification WG of the µITRON4.0 Specification Study Group, those involved in the ITRON Project, and those who directly or indirectly supported the process to develop the µITRON4.0 Specification. I would also appreciate your continuous support for the standardization activities of the ITRON Project.

June 1999

Hiroaki Takada

Secretary of the the µITRON4.0 Spefication Study Group
Department of Information and Computer Sciences,
Toyohashi University of Technology

Organization of the Specification Document

This document is the specification of the µITRON4.0 (or the µITRON4.0 Real-Time Kernel) C-Language API Specification. The version number of specification is printed on the cover and the top-right of each page.

The organization of this document is as follows.

In Chapter 1, a summary of the TRON Project and the ITRON Project and a design policy of ITRON Specifications are introduced. The position of the µITRON4.0 Specification is also described. This chapter describes the background information of the µITRON4.0 Specification and is not the main body of the µITRON4.0 Specification.

In Chapter 2, the common rule of the µITRON4.0 Specification and the software components that are standardized to be consistent with the µITRON4.0 Specification is described. In Chapter 3, the various concepts and the common definitions for various features of the µITRON4.0 Specification are shown. In Chapter 4, each feature of the µITRON4.0 Specification is described. In Chapter 5, the additional specifications are described.

In Chapter 6, the reference information, such as the maintenance of specification and reference documents, is described. In Chapter 7, the lists and other information that may be helpful in reading this specification is shown. These lists are the contents of Chapter 2 to Chapter 5, as seen from a different point of view. Chapter 6 and Chapter 7 are not the main body of the µITRON4.0 Specification.

Description Format of the Specification Document

The following description format is used in this specification document.

[Standard Profile]

The specifications of the Standard Profile of the μITRON4.0 Specification are described here. The scope of functionalities that the Standard Profile requires support, the rule that is not applied to the Standard Profile but is included in the functional descriptions of the services calls and static APIs which the Standard Profile requires to support, and the rule that is not described in the μITRON4.0 Specification but applied to the Standard Profile are described here.

[Supplemental Information]

Supplemental explanations of items difficult to understand to avoid misunderstanding are described here. This is not the main body of the μITRON4.0 Specification.

[Differences from the μITRON3.0 Specification]

The differences of the μITRON4.0 Specification from the μITRON3.0 Specification and their reasons are described here. The major differences and modifications from the μITRON3.0 Specification is mainly described, but not the additions or clarification made in the μITRON4.0 Specification. This is not the main body of the μITRON4.0 Specification.

[Rationale]

The reasons for the specification decision are described here, when further explanations are necessary. This is not the main body of the μITRON4.0 Specification.

The functional descriptions of service calls and static APIs in Chapter 4 uses the format described below.

The description of each service call or static API is started with the following header.

API Name API Description Profile

"API Name" is a service call or a static API name. "API description" is a simple statement about the functionality of this service call or static API. If [S] is placed at the "Profile" field, the Standard Profile requires support for the service call or static API.

[Static API]

This shows the description format of a static API in the system configuration file.

[C Language API]

This shows the invocation format of a service call from the C Language.

[Parameter]

This lists all of the parameters for this service call or static API. It also includes a simple description, the data type, and the name of each parameter.

[Return Parameter]

This lists all of the return parameters for this service call. It also includes a simple description, the data type, and the name of each return parameter.

[Error Code]

This lists all the main error codes that this service call returns. It also includes a simply description of the cause of each error code. However, the main error codes that many service calls may return due to the same cause are not described for each service call (see Section 2.1.6).

[Functional Description]

This describes the functionality of this service call or static API.

Italic characters within the names of service calls and constants represent other characters. For example, cre_*yyy* (*yyy* are italic characters) can be cre_tsk, cre_sem, cre_flg, and so on.

For some parameters, such as object attributes or service call operational modes where specific values are chosen, the following format is used:

[*x*]	*x* may or may not be specified
x | *y*	either *x* or *y* or both (bit-wise OR of *x* and *y*) may be specified
x || *y*	one of *x* and *y* must be specified

For example, ((TA_HLNG || TA_ASM) | [TA_ACT]) can take one of the following four values.

TA_HLNG
TA_ASM
(TA_HLNG | TA_ACT)
(TA_ASM | TA_ACT)

Table of Contents

Service Call Index

This is an index of the service calls defined in the μITRON4.0 Specification.

Static API Index

This is an index of the static APIs defined in the μITRON4.0 Specification.

Chapter 1 Background of µITRON4.0 Specification

1.1 TRON Project

1.1.1 What is the TRON Project?

TRON, which stands for "The Real-time Operating system Nucleus," is a project started by Dr. Sakamura of University of Tokyo in 1984 in an aim to establish an ideal computer architecture. Through collaboration between industrial world and universities, the TRON Project is aiming to produce an entirely new concept computer architecture.

In an effort to reconstruct the computer architecture, the TRON Project envisions the future to be a highly computerized society: a cyber society. In a cyber society, microcomputers are embedded in a majority of equipments, facilities, and tools that we encounter in our daily life. These devices are connected through a computer network and they work together in order to support our activities in various situations. Equipments with built-in computer and connected to the network are called "Intelligent Objects" while the overall system where intelligent objects are connected and work together is called "Highly Functional Distributed System" (HFDS). The realization of the HFDS is the most important goal of the TRON Project.

The TRON Project, divided into basic sub-projects and application sub-projects, is currently in progress. In the basic sub-projects, research is being conducted on the computer system, a component of HFDS. Specifically, the following sub-projects are currently in progress: ITRON (specifications of real-time OS for embedded systems and the related specifications), BTRON (specifications of OS for personal computers and workstations and the related specifications), CTRON (OS interface specification for communication control and information processing), and TRON HMI (standard guidelines for a human-machine interface of various products).

In the application sub-projects, analysis and evaluation are currently being conducted to solve problems associated with establishing a realistic application system in HFDS. A simulation of the future computerized society is also conducted as a basis for evaluation of the architecture developed in the basic sub-projects. The application sub-projects use the results of the basic sub-projects to solve the said problems while the basic sub-projects, in turn, make use of the feedback coming from the application sub-projects to further its research.

Toward the 21st Century

The TRON Project aims to establish an ideal computer architecture based on the technology of the 21st century. Our goal is to implement a top of the von Neumann-type architecture using VLSI technology, while giving utmost importance to real-time operations and cost performance. We adapt a new integrated design approach to a wide range of applications such as home electronics, industrial robots, personal computers, work stations, main frames, and private branch exchange (PBX).

Open Architecture

The basic policy of the TRON Project is to make the results of its research available through open specifications. Everyone can then freely develop and market his or her own products based on these specifications. This policy is essential in achieving the goal of developing HDFS. The TRON Association was established as the central organization to develop the TRON Specifications and to certify conformance to the specifications. Anyone can be a member of the TRON Association if they are in agreement with the concept of TRON and operate within the rules of the TRON Association.

Loose Standardization

The TRON Specifications define the interface of a computer, not the hardware or software it is founded on. It also defines the interface of the OS, but not the OS itself. The specifications are geared towards minimizing the development cost and upgrading the educational effects on users and programmers by implementing program and data compatibility. Thus, the TRON Association adapts the loose standardization, where only the design concept is defined. A developer can then freely implement a specific system that conforms to the design concept standard. Using a loose standard is a compromise between implementing the compatibility between HFDS components and allowing for the adaptation of new technologies.

The interface is defined in a layered structure, consisting of: the microprocessor instruction set, OS kernel, OS outer kernel, data formats, communication interface between objects, programmable interface, and the human-machine interface (HMI). With the layered structure of the specifications, various developers can independently implement different layers. Even in one system, different layers can be developed by different companies, and under free competition, same layers can be developed by different companies.

Future Compatibility

In order to realize the upward compatibilities in the future, the TRON Project is not affected by the compatibilities with the past. Many existing computer systems today are an enhancement of their early architectures. In other words, they are like houses renovated several times to make them larger. TRON, based on advanced VLSI technology, is an all-new architecture. TRON defines the standard data format, TAD (TRON Application Databus) to ensure compatibility for data that are transmitted between applications. The TAD format provides a means for TRON and other OS to coexist.

Standardization of Operation

Another goal of the TRON Project is to design computers anyone can operate, just like cars. Anyone can drive cars regardless of their manufacturer or model. The standardization of the HMI, just like in cars, is especially important for personal computers as it makes further knowledge unnecessary when a change or a revision in hardware and/or software components occur.

1.1.2 Basic Sub-Projects

ITRON (Industrial TRON) and JTRON

ITRON is an architecture for real-time operating systems (RTOS) for embedded systems. Details of the ITRON Specifications are provided in the following sections.

The JTRON Specification is a merger of the ITRON Specifications, which have been around for over 10 years, and the Java run-time environment, which excels in portability and network transparency. In application systems with the JTRON Specifications, it is easy to develop programs that uses the strengths of both ITRON and Java. More concretely, ITRON functions can be used to implement real-time control programs that have severe timing constraints, while Java functions can be used to manage GUI and other network-related functionalities. The JTRON Specifications have the following advantages. A real-time system with network functionalities can be constructed with the ITRON Specifications and Java. Components that need performance tuning can be coded with the RTOS's native code. On the other hand, components where portability is significant can be coded with the Java language. Thus, these components can be developed and debugged on a personal computer or a workstation.

The JTRON1.0 Specification was released in 1997, and the conforming products have already been released. The JTRON2.0 Specification strengthens the communication functionalities between the ITRON-Specification RTOS and the Java run-time environment.

BTRON (Business TRON)

BTRON refers to the architecture of personal computers and workstations that smoothly exchange information between humans and machines. It is important to guarantee data compatibility using a uniform HMI and TRON Application Database (TAD).

The main feature of the BTRON HMI is the GUI that supports keyboards and electronic pens as input devices. A touch panel can also be used instead of an electronic pen. BTRON is currently developing an HMI guideline that only supports pens.

TAD implements data compatibility between computers designed under the TRON architecture. It is a generic data format that can handle documents, graphics, and other real-time data (e.g. audio and video) for various environments.

BTRON1, BTRON2, BTRON3, and µBTRON Specifications have been released to meet our goals mentioned above. BTRON1 is designed to be implemented on a limited hardware resource. On the other hand, BTRON2 and BTRON3 are designed to make full use of the hardware resources of powerful computer systems. µBTRON is a BTRON Specification for PDAs and it provides power management function.

The BTRON1-Specification OS, which runs on a notebook type computer, was first released in 1991. TRON-Specification keyboards have also been developed for BTRON-Specification computers. They are designed for easier use and are less fatiguing than previous keyboard models. Electronic digitized pens are also used as a pointing device because they are more capable for handwritten character inputs and picture drawings compared to mice.

TRON-Specification keyboards were first sold in 1991. Now research is being conducted on the following areas: a new window system architecture for BTRON, TRON Application Control Language (TACL) which implements batch processing of graphical applications under BTRON, multi-media TAD specification, and TRON code that has a multi-language and multi-lingual support.

CTRON (Communication and Central TRON)

CTRON is an operating system interface that can be commonly applied to every exchange, communication, and information processing node on a communication network. Since the 1980s, which is said to be the start of the information society era, evaluation experiments have been conducted on CTRON interface specification, software portability and real time features.

The first version of CTRON Interface Specification was released in 1988. Since then, various works have been done to enhance and decrease the size of the specification, and in 1993, it was published as the new edition of “Original CTRON Specification Series.” The certification system of the CTRON-Specification OS was started in 1989, and up to this date more than 20 products have been certified.

From 1990 to 1992, an experiment on software portability was thoroughly executed. The objective of this experiment was to quantitatively evaluate the portability of the products conforming to the CTRON Interface Specifications. As a result, software portability was proven to be high, although some problems regarding software portability were also found. These problems have been reflected to the CTRON Specifications.

As mentioned above, CTRON was established as the basic software platform for communication networks in the 1990s. Now it is being considered as the core of communication networks essential to the multi-media generation of the 21st century.

TRON HMI

The HFDS is intended to help humans cope with daily lives by having multiple intelligent objects work together to provide support for humans. The TRON Project needs a uniform HMI in all HFDS environments. The purpose of this sub-project is to create an HMI guideline for intelligent objects, such as personal computers, electronic products, and automotive components.

The TRON HMI Guidelines describe the physical interactive parts that can be handled by users or used in applications such as buttons, switches, and handles. Enableware specification, and multi-language specifications are also available for a wide range of users. Enableware specification is for handicapped users while multi-language specification is for users who want to be able to control the computer in their own language. With an HMI made according to this guideline, a user can switch to other systems easily without worrying about system differences such as compatibility.

The result of this sub-project was presented as “TRON Human-Machine Interface Specifications.” In 1992 and 1993, the sub-project held competitions on HMI design in order to evaluate its usefulness.

1.1.3 Application Sub-Projects

Up to this date, experiments and research have been conducted on various application sub-projects, with the results taken as feedbacks to the basic sub-projects. Examples of application sub-projects are the TRON-Concept Intelligent House, the TRON-Concept Intelligent Building, and the TRON-Based Autotraffic Information System. The following sections introduce the four most recent application sub-projects being conducted.

Computer Augmented Environment

The computer augmented environment refers to an environment where computers are embedded in every machine, and each machine, in turn, is connected to a network, thus expanding the functionalities of the real environment. It is being studied by many researchers throughout the world. The term HFDS discussed above actually refers to the computer augmented environment, and its construction is the TRON Project’s final objective.

In order to realize the computer augmented environment, we are currently developing a “Computer Augmented Environment Control Script” designed to handle the control embedded devices from personal computers and servers.

Multi-Media Network Service Platform (MNP)

The rapid spread of the internet and intranet provides an opportunity for networked multi-media services.

Since 1994, much work has been done to adapt CTRON to multi-media network services. CTRON is focusing on the usage flexible resources and implementation of real-time control functions on a network and its peripherals (such as nodes and routers for gateway functions, servers and multi-media terminals.) New OS interface rules have been added and technical problems regarding control functions required by the focus mentioned above are being continuously examined since 1994.

Digital Museum

The digital museum is a futuristic museum that uses digital technology in every operation phase, including exhibits and presentations. The digital museum is not a virtual exhibit on the web. Virtual exhibition, itself, is a part of a digital museum. The concept of the digital museum is to extend and strengthened the real space of a physical museum using cyberspace tools, such as computers and the Internet, thereby overcoming the limitations imposed on a real museum and at the same time increasing its appeal.

The digital museum is an example of an HFDS application. The required computer technology in constructing the digital museum is actually BTRON's goal. This fact shows the exclusiveness of BTRON technology and at the same time indicates that the direction of the hypermedia technology developed under BTRON sub-project is correct.

Distributed Software Platform for Information Home Electronics

The digitalization of home electronics and the use of home networking have rapidly advanced in the recent years while software is needed to control information home electronics are getting more and more complex. On the other hand, much shorter time for the development of devices are being imposed, thus heightening the need for software platforms to increase software development efficiency.

Middleware groups have been built to connect information home electronics to networks, using μITRON-Specification RTOS. The μITRON Specifications provide a foundation for efficient software development while ensuring the connectivity and operability of embedded products.

1.2 History and Current Status of the ITRON Specifications

1.2.1 Current State and Features of Embedded System

With the progress in microprocessor technology, the range of applications in which embedded systems are practically used has significantly increased. During the early days, embedded systems were mainly limited to industrial applications such as production line control. Now, embedded systems are rapidly spreading to office electronics, communication products, and most recently, to consumer products like automobiles, audio/video systems, televisions, cellular phones, electronic instruments, games, laundry machines, air conditioners, and lighting systems. The term embedded system now applies to most of the electronic products we encounter in our daily lives.

With the increased range of applications for embedded systems, the functions that these systems must perform become more complex. In addition, the recent trend towards digitalization and the increase in number of software-implemented process on highly functional microprocessors makes embedded systems more significant.

In general, small-scale embedded systems, usually consumer products, are produced in large quantities compared to large-scale embedded systems typically found in industrial products, making the cost per product comparatively cheaper. While decreasing the development costs for large embedded systems is given importance, decreasing the manufacturing costs of small-scale embedded systems is significant. In particular, because of the tight competition on product development, attempts are made to shorten the development time of consumer products. In addition, sold softwares are rarely redesigned, which results in a very short life cycle for system development.

In most small-scale embedded systems, the core processor, ROM and RAM, general I/O devices, and some other devices are all in a chip called MCU (Micro Controller Unit, sometime called “one chip micro processor.”) Since the development cost of the final product is to be kept as low as possible, hardware resources on a MCU, especially the memory, are very limited. This limitation becomes a problem when developing softwares on a MCU. The highly efficient MCU has various kinds of processors optimized and designed for applications.

In small scale embedded systems, improving software productivity is important in handling largely scaled and highly complex softwares. It is also significant in reducing the software development time. It is often to use a high-level language like C, and an RTOS, like a µITRON-Specification RTOS.

1.2.2 Requirements for RTOS on Embedded System

To keep up with the progress of high performance microprocessors technology, it is very important for embedded systems to be cost-effective, especially since they are now widely applied to consumer products. Also the number of software engineers working on RTOS is also increasing as embedded systems are being applied to more and more areas, making their education a lot more significant.

In a survey conducted by the TRON association every year from 1996, the survey shows the greatest problem encountered by most engineers using an RTOS in an embedded system is regarding education and standardization. The survey shows that there are very few engineers who can handle RTOS and that the specifications of different operating systems are so large that switching to another OS would take a lot of work. The survey also shows that the OS size and resources are too large, and most of its features and functions do not meet actual requirements, leading to problems in matching an OS with an application.

The TRON Project, giving importance to education from aspect that standardization of concept and technical-term, has decided to provide a standardized RTOS specification that can easily be applied in many embedded systems.

The most difficult problem encountered in providing a standardized RTOS specification for embedded systems is finding the balance between providing the highest performance that the hardware allows and upgrading software development productivity. On MCU based systems with tight limitations on hardware resources, reaching the maximum hardware performance will only be achieved by carefully selecting the appropriate RTOS. On the other hand, improving software development productivity involves increasing the abstraction of OS services and guaranteeing software portability regardless of the hardware in use would increase the gap between OS services and the hardware architecture. This gap would cause significant overhead and getting a high performance from hardware would be a lot more difficult.

The compromise between these two goals highly depends on the performance of embedded products. Particularly, it is meaningless to lower the runtime performance of small scale embedded systems just to keep the final product's cost low and improve its portability. On the contrary, since large scale systems are often recycled, portability is a very important issue. The optimal solution to this problem is not well defined and the optimal balance point changes with the progress of microprocessors.

Small and large scale embedded systems often require different RTOS features. Small scale systems would often suffer decreased performance and increased program size from using an RTOS with many high-level features that are really unnecessary. On the other hand, an OS with many high-level features is useful for large-scale embedded systems, as it helps improve software development productivity.

As seen from above, the requirements for an RTOS differ depending on the scale and the necessary features of each embedded system. It would be possible to define an RTOS specification for each application scale or required feature sets. However, in considering the education of software engineers, the software circulation, and the support for development tools, defining a scalable OS specification that can adapt to the needs of a variety of embedded systems, would be very useful.

The following is a summary of the requirements for the specification of an RTOS on embedded systems:

- To be able to get the maximum performance from the hardware.
- To be useful in increasing productivity for software.
- To be able to adapt to any system scale (scalability).

In addition to the above technical requirements, it is also important that the specification be open. Because embedded systems are involved in all the electronics products that we encounter daily, it is necessary not only to make the specification available to every one, but also to make it royalty free so that anyone can implement and sell products based on the specification.

1.2.3 Current Status of the ITRON Specifications

The ITRON Project started in 1984, and it has developed and released a series of ITRON Real-Time Kernel Specifications. The project gave utmost importance to the standardization of kernel specifications because small scale systems often only use just the kernel functions.

The first ITRON specification was developed in 1987 as the ITRON1 Specification. Many real-time kernels were developed based on the ITRON1 Specification, and they served to be useful in verifying the specification's usability. Later, in 1989, the ITRON Project released two specifications: the μITRON Specification (ver. 2.0) and the ITRON2 Specification. The μITRON Specification is for small systems on an 8 or 16-bit MCUs. One of its characteristics is limited the kernel functionality. The ITRON2 Specification, on the other hand, is designed for larger systems on 32-bit MCUs. The μITRON Specifications have been implemented on many different MCUs with limited memory and limited computational resources. It is also used on a wide variety of embedded systems and it provides practical functionality without large memory requirements. In fact, μITRON-Specification kernels have been developed on most major MCUs used in embedded systems.

In order to apply the μITRON Specifications to a wide range of fields, functionality and performance are necessary. Even though the μITRON Specifications was not designed for 32-bit processors, the μITRON-Specification kernel is now being implemented on 32-bit MCUs since the kernel does not consume significant memory. Because of this, the specification was revised to make it scalable on MCUs ranging from 8-bits to 32-bits. The revised edition was the μITRON3.0 Specification, released in 1993. The μITRON3.0 Specification includes connection functions that allow a single embedded system to be implemented over a network. IEEE CS Press published the English version of the μITRON3.0 Specification under the title "μITRON3.0: An Open and Portable Real-Time Operating System for Embedded Systems."

At present, there are approximately 50 ITRON real-time kernel products for 35 processors registered to the TRON association. There is also a U. S. software vendor that has developed a μITRON-Specification kernel. Since the μITRON-Specification kernel is small and is easy to implement, many users have developed their own versions for in-house use. There are also several implementations that besides products, and some versions of the μITRON kernel are distributed as free software.

The reason that μITRON kernels are used in so many instances is that it supports a wide range of applications. Table 1-1 shows examples of some devices that use ITRON kernels. From the survey mentioned in the previous section, the ITRON Specifications are used often in consumer products and it has become the standard among industrial companies. Many companies develop their own ITRON-Specification kernel, which indicates that the ITRON Specifications are truly open standards.

Table 1-1. Major Fields where ITRON-Specification Kernels are Applied

Audio/Visual Equipment, Home Appliance	TVs, VCRs, digital cameras, STBs, audio components, microwave ovens, rice cookers, air-conditioners, washing machines
Personal Information Appliance, Entertainment/Education	PDAs (Personal Digital Assistants), personal organizers, car navigation systems, game gear, electronic musical instruments
PC Peripheral, Office Equipment	printers, scanners, disk drives, CD-ROM drives, copiers, FAX, word processors
Communication Equipment	answer phones, ISDN telephones, cellular phones, PCS terminals, ATM switches, broadcasting equipment, wireless systems, satellites
Transportation, Industrial Control, and Others	automobiles, plant control, industrial robots, elevators, vending machines, medical equipment, data terminals

In addition to the real-time kernel specifications, the ITRON Project also provides the ITRON/FILE Specification that provides file management features compatible with the BTRON-Specification file system.

Many widely used products use processors with the ITRON Real-Time Kernel Specification. The µITRON-Specification kernel has been especially useful on MCUs, which were not previously used on RTOS due to memory and speed restrictions. The µITRON Specification brings us closer to achieving the standard real-time kernel specification possible.

The object of standardization is now widened to include, not just the kernel, but also software components, development tools, and related specifications. Also, research and standardization on each application field is in progress (see Section 1.4.1). The research and studies conducted by the TRON Project are all directed to realizing its ultimate goal: the HFDS.

1.3 ITRON Specification Design Policy

The following policies are adapted in designing the ITRON Specifications. These policies satisfy the requirements for an RTOS given in Section 1.2.2.

- Excessive hardware virtualization should be avoided in order to increase adaptability to the hardware.

  In order to maximize the performance of the hardware and thus, acquire high real-time efficiency, excessive hardware virtualization should be avoided. The phrase "adaptability to hardware" refers to improving the performance of the whole system by modifying the RTOS specifications and/or RTOS internal implementation according to the hardware's performance and characteristics.

  More specifically in the ITRON Specifications, items that should be standardized regardless of the hardware structure are clearly divided from the items that can be optimized according to the hardware's performance and characteristics. Standardized items includes task-scheduling rules, system call names, system call functionalities, names, order, and meanings of system call parameters, and names and meanings of error codes. On the other hand, items that would cause a decline in performance are not forcibly standardized, instead, standardization and virtualization are purposely avoided. For instance, bit width of parameters and methods for invoking interrupt handlers are decided on each implementation.

- Adaptability to applications should be considered.

  Adaptability to application refers to the approach to improve the over all system performance by modifying the kernel specifications and internal implementation methods in response to the kernel functionalities performance required by applications. Since the object code for the OS is created for each application, adaptability to applications approach works well in embedded systems.

  The specification is designed in such a way that each kernel function is kept independent to each other as possible so that only the required function for each application are actually used. Providing a single functionality to each system call makes incorporating of only the required functions easier. Most μITRON-specification kernels are provided as libraries and only the required modules are extracted and linked with application programs.

- Education of software engineers should be given importance.

  Compatibility and portability are not of a great concern to softwares developed for small embedded systems because the software is not likely to be reused. However, standardizing the kernel specification is more important because it helps to educate software engineers. It also make communications between software engineers easier because by unifying technical terms and concepts.

  In the ITRON Specification, the education of software engineers is given importance. By standardization, an engineer can widely apply what he learns once. The usage of terms and naming of system calls, for example, are made as consistent as possible. Educational text books for engineers are also in progress.

- A series of specifications should be developed and support levels should be defined in a specification.

  In order for applications to adapt to various hardwares, a series of specifications that allow different scalable levels of support are created. The series of real-time kernel specifications made up to this date includes µITRON Specification (Ver. 2.0) for 8 to 16 bit MCUs and ITRON2 for 32 bit processors. With these specifications, the user can scale each functionality as needed and include only those functionalities when implementing the kernel. The µITRON3.0 specification separates the systems calls into different levels of support to cover both small-scale and big-scale processors within one specification.

  Specifications for distributed systems over a network, and multi-processor systems are also being considered for standardization under the ITRON Specification series.

- Various functionalities should be provided.

  The ITRON Specifications provide a large set of primitives with different properties to cover a wide range of functionality instead of limiting the number of primitives. Using the primitives according to the natures and characteristics of the application and hardwares, improve performance during execution and makes program coding easier.

The common concept among the above design policies is “loose standardization.” Loose standardization means that some parts of the specification that would reduce the hardware performance are not forcibly standardized and are left to the developer to implement on hardware and/or application. With loose standardization, maximum performance for various hardware platforms is achieved as shown in Figure 1-1.

1.4 Position of the µITRON4.0 Specification

1.4.1 Second Phase Standardization Activities of the ITRON Project

As mentioned previously, the ITRON Project has been focusing on standardization of real-time kernel specifications. As the embedded systems become larger and more complex, the need for standardization on the surrounding environments of the real-time kernel is increasing. In 1996, the ITRON Project started its second phase: expanding standardization from kernel specification to the kernel’s related specifications, especially on software components for embedded systems.

In standardizing software components, not only the conditions for advancing the development and distribution of software components but also the interface for different

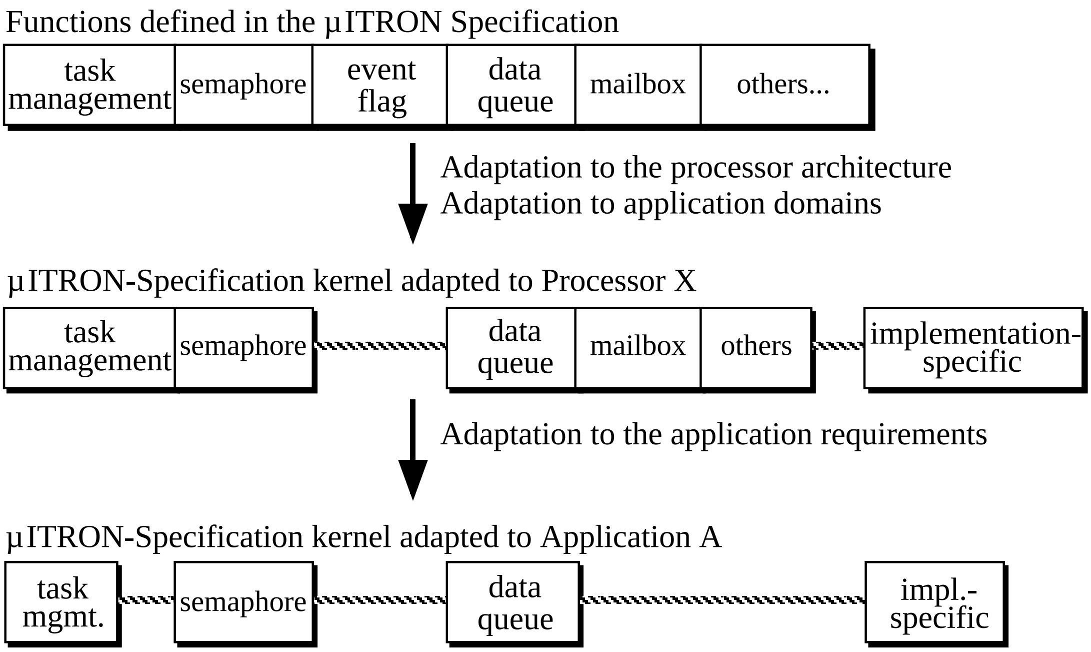


Figure 1-1. Adaptation according the µITRON Specifications

fields are being considered.

The following two issues are being discussed to prepare the conditions for advancing the development and distribution of the components. The first problem is regarding the distribution of software components. The difference in implementation among µITRON kernels makes it difficult to ensure the distribution of software components. To solve this problem, it is necessary to raise the level of kernel standardization while keeping the advantages of loose standardization. The second problem is regarding the support for software components with real-time capability. Many software components are required to have real-time capabilities and a framework is needed to allow the coexistence of software components and application while satisfying software components' real-time restrictions. The framework also allows multiple software components to be used together.

The discussion results regarding these two problems are reflected in the µITRON4.0 Specification. A standard method for designing embedded systems with real-time kernel is also proposed. A guideline for designing applications supporting software components with hard real-time capability is being created.

Standardization of software component interface in every field currently in progress includes API (Application Program Interface) for TCP/IP protocol stacks and standard interface for Java execution environments.

The TCP/IP protocol stack has taken an increasing significance in the field of embedded systems, recently. Though the socket interface is in wide use today as a TCP/IP API, it is not appropriate for embedded systems (particularly small-scale ones) because of such problems as its large overhead and the necessity of dynamic memory management within the protocol stack. The ITRON TCP/IP API Specification, which is a standard TCP/IP API for embedded systems, has been designed to solve these problems of the socket interface and to enable a compact and efficient implementation of the TCP/IP protocol stack. The ITRON TCP/IP API Specification has been published on May, 1998.

Java technology is also drawing interest these days. A practical approach for applying Java technology to embedded real-time systems is to implement the Java runtime environment on an ITRON-specification kernel. Then, build an application system whereby the parts for which Java is best suited are implemented as Java programs, and the parts taking advantage of the ITRON-specification kernel strengths are implemented as ITRON tasks. A key issue here is the standardization of the communication interface between Java programs and ITRON tasks. The JTRON2.0 Specification has been designed to define this interface standard and published on Oct., 1998.

Besides software component support, defining the requirements for ITRON kernels designed for automotive control and gathering proposals for the standard specification were also conducted. The results are included in the µITRON4.0 Specification.

Works on standardizing interface between ITRON-Specification kernels and debugging environments, and guidelines for designing device drivers are currently in progress. Furthermore, C++ language bindings for the ITRON kernel are also being surveyed.

1.4.2 Necessity of the µITRON4.0 Specification

The need for reconsidering the real-time kernel specification arose during the ITRON Project's second phase mentioned in the last section and as a result, the µITRON4.0 Specification was created. This specification is considered as the 4th generation of ITRON Specifications. The four main reasons why it was necessary to design the µITRON4.0 Specification is outlined below.

(a) To improve software portability

Embedded software continues to grow in complexity and size. The need for applications to easily switch to different kernels is increasing. Portability of softwares developed on an ITRON kernel is also an important issue in the distribution of software components.

(b) To add functionality for supporting software components

The original µITRON Specifications left out some functionality to create software components that are intended for the market. For example, the functionality to find the context where a service routine of a software component is called was only available on the extension level.

(c) To include new requirements and results of studies

From November 1996 to March 1998 a research group on hard real-time support studied functionalities needed by a real-time kernel to make it easier to build a hard real-time system. The RTOS automotive application technique committee, from June 1997 to March 1998, sorted out the requirements for real time kernels on automotive control applications. The results of these new requirements and studies must be included in real time kernel specification.

(d) To include enhancements allowed by improved semi-conductor technology

Six years after the release of the µITRON3.0 Specification, the semi-conductor technology has dramatically progressed and so is the performance of embedded processors. The available memory size on processors has also drastically increased. Some useful kernel functions that were left pending on the release of the µITRON3.0 Specification due to their overhead, can now be implemented with the current technology.

1.4.3 Introduction of the Standard Profile

In order to improve software portability, the set of functions required for implementation and the functional specification of each service call should be strictly regulated. In other words, the grade of specification standardization must be made stronger.

The standardization of µITRON Specifications has been done along the "loose standardization" policy which gives more importance to adaptability on hardwares and processors rather than software portability by reducing overheads and memory size during execution time. "Loose standardization" policy has made µITRON Specifications scalable and acceptable across a wide range of processors ranging from 8bits to 64bits. This is one of the important reasons why the µITRON Specifications are widely accepted. However, improving software portability and realizing scalability have many contradicting aspects. It is difficult to realize both requirements at the same time within one specification.

To address the issue of portability while maintaining the "loose standardization" policy, the µITRON4.0 Specification strictly defines the set of standard functions and their specifications. This set of standard functions is called the "Standard Profile." A large-scale system was assumed when defining the Standard Profile for the µITRON4.0 Kernel Specification. This is because larger systems require a more portable software.

Defining the Standard Profile leads to encouraging the building of softwares using only functions provided by the Standard Profile, in cases where the portability of software components is significant. It also leads to encouraging the implementation of kernels, where the portability of software components are important, based on the Standard Profile.

Within the Standard Profile, the specification is made to maximize software portability while maintaining scalability. As an example, a mechanism for improving the portability of interrupt handlers while keeping overheads small, has been introduced. Previous µITRON Specifications did not provide a way to maintain portability in prohibiting the nesting of higher priority interrupts from within an interrupt handler. However, the μITRON4.0 Specification does.

In realizing scalability, service calls are made as independent to each other as possible, and many sets of functions are made available, but only the necessary functions are actually linked using the library link mechanism. This method is the same as that of previous μITRON Specifications. When it is difficult to link only the necessary functions using the library link mechanism, then the kernel is supposed to provide, only the necessary primitives required to support more complex functions. This enables the support of complex function without modifying the kernel, while minimizing the overhead in an application requiring no complex functions.

The Standard Profile assumes the following system image.

- The Standard Profile assumes the following system image.
- High-end 16 or 32-bit processor is used.
- The kernel code size is about 10 to 20KB when all functions are included.
- The whole system is linked into one module.
- The kernel object is statically generated.

Since the whole system is linked into a single module, service calls are invoked using subroutine calls. The system does not have any particular protection mechanism.

The functions to be supported in Standard Profile includes all the level S functions (with modifications and expansions in some functions) and a part of level E functions (such as service calls with timeout, fixed-sized memory pool, cyclic handlers with specification sorted out) of the μITRON3.0 Specification, and newly introduced functions (task exception handling, data queues, system state reference function, and so on). The static API used to state object creation information (to be described later) is also supported.

1.4.4 Realization of a Wider Scalability

As described in the previous sections, the μITRON4.0 Specification maintains a policy of "loose standardization" and at the same time aims to provide a wider scalability than the previous ones.

It defines a minimum function set that can be made more compact than the previous μITRON Specifications and more adaptable to small systems. Specifically the waiting state that was mandatory in the μITRON3.0 is no longer required. It is, however, replaced by the dormant state, which is mandatory. A kernel without the waiting state allows tasks to operate within the same stack space. This reduces required memory area and overhead on context switches.

In order to support the requirements over the Standard Profile, the full set of μITRON4.0 Specification provides more functions than the full set of the previous specifications. Specifically it includes almost all the functions of the μITRON3.0

Specification excluding the connection functions. Newly introduced functions in µITRON4.0 Specification includes: the new functions in the Standard Profile (task exception handling, data queue, system state reference function), object creation functions for automatic assignment of ID number, interrupt service routine functions enabling interrupt handling written while keeping portability, mutex to support priority inheritance/ceiling protocols, overrun handler to detect the time left assigned to a task. The full set of µITRON4.0 Specification is no less than the full set of ITRON2 Specification in terms of functionality.

In addition to the Standard Profile, an "Automotive Control Profile" is also defined. The Automotive Control Profile targets automotive control applications. It is also considered as a function set that increases the software portability for systems smaller than those targeted by the Standard Profile. Specifically, Standard Profile functionalities, such as functions with timeouts, suspended states, task exception handling, mail boxes, and fixed-sized memory pools are unnecessary and therefore was omitted. On the other hands, a task called a restricted task, is uniquely defined in the Automotive Control Profile. Restricted tasks do not enter the waiting state so restricted tasks with equal priority can share the same stack area, reducing memory use. Unless there is no dependency on errors occurring from invoking a service call that enters the waiting state, restricted tasks can be replaced by normal tasks, and the resulting behavior does not change. The Automotive Control Profile is backward compatible with the Standard Profile, even with the specific functionality of restricted tasks.

Figure 1-2 illustrates the µITRON4.0 supported function levels relative to the µITRON3.0 Specification. Compared to previous specifications, the µITRON4.0 Specification is more applicable to smaller and larger systems.

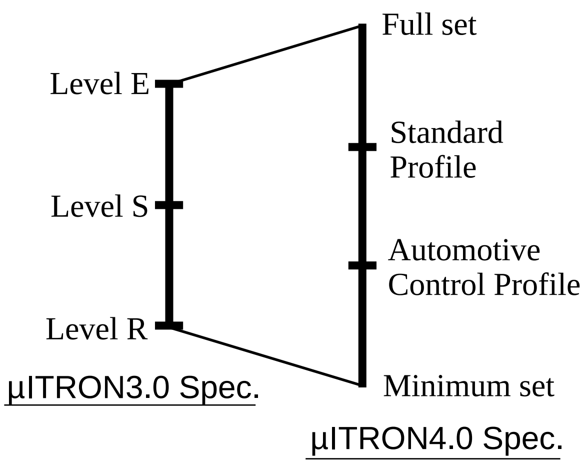


Figure 1-2. Function Levels of µITRON4.0 Compared to µITRON3.0

1.4.5 New Functions in the µITRON4.0 Specification

New functions were added to the µITRON4.0 Specification are described below.

Exception Handling Functions

The µITRON4.0 Specification defines the frame for exception handling, which was left as implementation-dependent under the previous µITRON Specifications.

When the processor detects an exception condition, the processor starts a CPU exception handler. CPU exception handlers can be defined for each type of exception in the application. Since a CPU exception handler is global in the overall system, it is possible to check the context or the situation where an exception occurs, from within the Task exception handling function is like a simplified version of a UNIX signal function and is similar to the ITRON2 compulsion exception. The following is a list of typical applications using task exception handling functions:

- Signal CPU exception, such as division by zero error, to a task.
- Send a task termination request to another task.
- Notify a task that the deadline has been reached.

Functionalities defined by the µITRON4.0 Specification for exception handling are designed so that they can be used as primitives in implementing more complicated exception handlers.

Data Queues

A data queue is a mechanism to communicate with a single word data message. The µITRON3.0 Specification permitted use of either a linked list or a ring buffer to implement a mailbox. However, in the µITRON4.0 Specification, the implementation of a mailbox is limited to a linked list. In addition, a data queue, which is equivalent to a mailbox implemented with a ring buffer, is introduced as a separate object.

Data queue feature was strongly required by the Automotive Control Profile and was at first, introduced as a unique feature of the Automotive Control Profile. However, since data queues are useful to other application areas and programs not requiring data queues can be implemented without linking them to data queues, the date queue feature was included in the Standard Profile.

System State Reference Functions

When creating software components assuming they are called by applications developed elsewhere, each service routine in each software component should work regardless of the context where it is called. However, in the µITRON3.0 Specification, only the level E system call, ref_sys, was able to look at the present system state. Many implementations do not support ref_sys, and even in cases where ref_sys is supported, the large overhead caused by reference to unnecessary information was a problem.

In response to this problem, 5 new service calls of the form sns_*yyy* have been introduced in the µITRON4.0 Specification. These service calls can refer the current state of the system with small overheads. They can be invoked from any context and will return a Boolean value (They will never return an error.) As an example, it is possible to check, without worrying about overheads, whether a service call that enters waiting state can be invoked or not.

Also, to handle processing that requires mutual exclusion, these service calls facilitate the locking CPU (or disabling dispatch) temporarily, and then restore the system back to the original state after the processing is finished. The µITRON3.0 Specification has no feature to restore the system to its original state once loc_cpu was invoked during the dispatching disabled state. In the µITRON4.0 Specification, on the other hand, dispatching disabled state and CPU locked state are independent from each other so no problems arise in locking the CPU.

Object Creation Functions for Automatic ID Number Assignment

In the µITRON3.0 Specification, the ID number must be provided in creating an object dynamically. In large-scale systems, managing unused ID numbers is tedious. In the µITRON4.0 Specification, service calls are introduced to create an object using the ID number assigned by the kernel instead of the ID number specified by the application. The service calls return the assigned ID number.

Interrupt Service Routines

The interrupt handling architecture depends on processors and systems, and is a difficult part to standardize. The previous µITRON Specifications did not standardize the coding of interrupt handlers and was determined and optimized for each processor and system. However in order to improve portability of device drivers, a method to write portable device drivers is required.

The µITRON4.0 Specification introduced an interrupt service routine functions to write interrupt handling while preserving the portability as well as the interrupt handler functions in the previous specifications. The specification of interrupt service routine is being designed with the goal of writing interrupt routines that depend only on interrupt generating devices.

Mutexes

Priority inheritance protocols and priority ceiling protocols are necessary to prevent priority inversions in a system with severe real-time constraints. Mutex is a mutual exclusion mechanism that supports priority inheritance protocols and priority ceiling protocols. It is a new feature of µITRON4.0 Specification. The mutex feature in the real-time extension of POSIX real-time was referred to when designing the mutex for the µITRON4.0 Specification.

Overrun Handler

Overrun handler is another feature required in building a system with severe real-time constraints. Overrun handler detects whether the amount of processing time assigned to a task has been used up.

The simplest method to detect that a timing constraint has not been met in a system is by checking if the processing does not finish by the designated deadline. This can be done using an alarm handler. However, this method do not prevent higher priority tasks from continuing to run until its deadline, and as a chain reaction result, lower priority tasks may not meet their deadlines. To solve this problem, a mechanism to detect when a task has used up given amount of time is required.

Standard Configuration Method

The Standard Profile assumes that the kernel objects, such as tasks and semaphores, are created statically. In order to port the application software written on a kernel conforming to the Standard Profile to another conforming kernel, in addition to the application program itself, object creation information must also be ported to the new kernel.

Previous μITRON Specifications did not standardize descriptions for the creation of information in the kernel causing incompatibility in between kernels. For example, one product may write the object information using C data structures, while another product may write the object information statically through a GUI configuration utility. When porting a large scaled application to another kernel under such conditions, the amount of work on the porting of creation information can no longer be ignored. Although the actual work of rewriting itself is not big, attention should be paid to the fact that the amount of time required to learn a different way of writing for different products must be included in the total amount of work.

The μITRON4.0 Specification standardizes the coding of object creation information and the way to configure the kernel or software components based on those information. The method of writing object creation information in the system configuration file is called static API. The names of static APIs are the same as names of the service calls with the corresponding function, but they are written in upper case letter. Static APIs and service calls share the same parameters except that each element of a packet is written within “{” and “}” instead of passing a pointer to the packet. Because of this, learning either the static API or the service call means learning the other. This is intended for educational purposes.

The configurator which processes static APIs must have a function to automatically assign ID number to the object with no ID number given. This allows omission of handling of automatic ID assignment, even when building an application with separately developed modules and is very useful for large scale application development.

Static APIs for software components as well as static APIs for the kernel can be described in one system configuration file. This is another feature of the configuration method of the µITRON4.0 Specification. By having the system configuration file processed by the software component configurator first, and then by the kernel configurator, complicated situations, such as the case where software components require different kernel objects on their configuration, can be handled.

In addition to the new features introduced above, the µITRON4.0 Specification also reduces implementation-dependency by defining those items that were left ambiguous or implementation-dependent in each service call function under the µITRON3.0 Specification in order to improve the software portability. Also many improvements have been made over the µITRON3.0 Specification, such as sorting out terms and concepts, sorting out data types of parameters, sorting out error codes, reassigning function codes to service calls, standardizing constants and macros to retrieve kernel configuration, and standardizing system initialization process.

Chapter 2 ITRON General Concepts, Rule, and Guidelines

The ITRON general concepts, rules, and guidelines stated in this chapter are common to the µITRON4.0 Specification and the software component specifications standardized to be consistent with µITRON4.0. These specifications are referred to as the ITRON Specifications. In the ITRON general concepts, rules and guidelines, the "kernel specification" refers to the µITRON4.0 Specification and the "Standard Profile" refers to the Standard Profile of the µITRON4.0 Specification.

[Supplemental Information]

As mentioned above, the concepts, rules, and guidelines in this chapter are applicable to software component specifications as well. However, to make the µITRON4.0 Specification more understandable, we refer to certain areas specific to the µITRON4.0 Specification and its Standard Profile when necessary.

2.1 ITRON General Concepts

2.1.1 Terminologies

Terminologies used in this specification are defined below.

- Implementation-Defined: Items that are covered in the functional description of the ITRON Specifications but are not standardized by the specifications. All implementation-defined items should be defined and described by the implementation's documentation, such as the product manuals. The portability of any part of an application program that depends on implementation-defined items is not guaranteed.
- Implementation-Dependent: Items covered in the functional description of the ITRON Specifications, but whose behavior varies depending on the implementation and on the system operating conditions. The specifications do not guarantee the behavior of an application program that relies on implementation-dependent items.
- Undefined: Situations with no guaranteed behavior. That is, a system failure might occur in any undefined situation. Items not mentioned in the specifications are generally undefined. There is no guarantee in the specification for the behavior of an application program that generates an undefined situation.
- Implementation-Specific: Functionalities, which are beyond the scope of the ITRON Specifications and are defined by the implementation.

[Supplemental Information]

Features defined by the implementation do not need to be internally consistent within the implementation and may vary according to the kernel or software component configurations. In the case where variations in feature definitions exists due to the kernel or software system configuration, implementation documents such as product manuals, should describe the feature definitions for each configuration, as well the steps in configuring the kernel or the software component.

2.1.2 Elements of an API

An API (Application Program Interface) is a method used by an application program to interface to the kernel or a software component. An API consists of the following elements:

(A) Service Calls

The interface used by an application program to call a kernel or a software component is referred to as a service call. The ITRON Specifications standardize the names and functions of service calls, as well as the types, orders, names, and data types of their parameters and return parameters.

In a C language API, a service call is defined as a function call. However, it may be implemented in other forms such as a preprocessor macro as long as it has the same functionality.

[Differences from the µITRON3.0 Specification]

In the µITRON3.0 Specification, the service call concept was referred to as a system call. The concept name has changed to service call in order to include software components as well as kernel functionalities. The term system call may still be used to refer to a kernel service call.

(B) Callbacks

The interface used by a software component to call a routine registered by an application program is referred to as a callback. The registered routine is called a callback routine. The ITRON Specifications standardize the names and functionality of callback routines, as well as the types, order, names, and data types of their parameters and return parameters.

The context in which a callback routine is executed is defined in each software component specification.

[Supplemental Information]

Callbacks are not used in the kernel specification.

(C) Static APIs

Static API refers to the interface used in both determining the kernel or software component configuration and defining the initial states of objects within a system configuration file. The ITRON Specifications standardize the names and functionalities of static APIs as well as the types and order of their parameters.

Service calls, such as those used to register objects, may have a corresponding static API. The functionality of a static API is equivalent to executing the corresponding service calls during system initialization, in the order listed in the system configuration file. Some static APIs, like the ITRON general static APIs commonly used by kernel and software components, do not correspond to any service call at all.

(D) Parameters and Return Parameters

Parameters are data passed to service calls, callback routines, and static APIs. Return parameters, on the other hand, are data returned by service calls or callback routines. The ITRON Specifications standardize the names and data types of parameters and return parameters.

In a C language API, the return parameters, except the return value of a function, are returned either through a pointer passed as an argument to a C language function, or as a data structure containing multiple parameters or return parameters. This type of structure is called a packet. The pointer that points to the area holding the return parameters is not listed as a parameter. In the case where a pointer is pointing to a single return parameter, that pointer is not listed as a parameter, while a pointer to a packet, on the other hand, is listed as a parameter. In a C language API, an argument pointing to an area holding a certain return parameter is named by prefixing the return parameter's name with "p_." If the return parameter's name starts with "pk_," the pointer to the return parameter starts with "ppk_." When parameters are too large to pass as an argument, a pointer to the data area holding the parameter may be passed instead. The naming convention for return parameters applies for parameters as well.

As a general rule, the data areas used to hold the parameters and return parameters of a service call can be reused by the application once the service call returns. Also, data areas used to hold callback routine parameters and return parameters for a software component can be reused by the software component once the callback routine returns. Exceptions to these rules are explicitly mentioned in the functional descriptions of service calls and callbacks.

[Rationale]

Standardizing the argument and return value names of functions is actually not necessary since they do not affect any kernel or software component API functionality. However, the names of C language function arguments and function return values are standardized in the ITRON Specifications because they are used frequently throughout the specification and product manuals.

(E) Data Types

The ITRON Specifications standardize the names and meanings of parameter and return parameter data types. Some data type definitions are standardized in the ITRON Specifications.

(F) Constants

The ITRON Specifications standardize the names, meanings, and values of the constants used as parameters, return parameters, and function codes for service calls. In a C language API, constants are defined using preprocessor macros.

(G) Macros

A macro is an interface to convert values which are not bound to the system state without calling the kernel or software components. The ITRON Specifications standardize the names and meanings of macros. In a C language API, macros are defined using preprocessor macros.

(H) Header Files

There is one or more header files for each kernel and each software component containing declarations of service calls and definitions of data types, constants, and macros. The ITRON Specifications standardize the names of these header files. If there are more than one header file, the standardization also covers which header files contain which declarations and definitions.

A header file containing the definitions of data types, constants, and macros specified in the ITRON General Definitions section should be included in header files prepared for each kernel and software component.

The configurator automatically assigning object ID numbers generates an automatic assignment header file to contain the generated IDs. ITRON Specifications standardize the names of these header files.

The header files standardized in the ITRON Specifications can be divided into more than one file depending on the implementation. Care should be taken so that no error arises even when the same header file is included multiple of times.

[Supplemental Information]

To prevent errors due to multiple inclusion of the same header file, define a specific header identifier, for instance "KERNEL_H_," as a preprocessor macro ("#define _KERNEL_H_") at the top of the header file, and then enclose the whole header file with "#ifndef _KERNEL_H_" and "#endif."

2.1.3 Object ID Numbers and Object Numbers

The resources on which a kernel or a software component operates are generally referred to as objects. Objects of each type are uniquely identified by numbers. In the case where only a kernel or a software component API uses the object identifier and the application is allowed to freely assign numbers, the identifier numbers are called ID numbers. On the other hand, identifier numbers are called object numbers if they are assigned according to an internal or external condition of the kernel or a software component.

Objects identified by ID numbers are registered to the kernel or a software component when the application creates them. Objects identified with object numbers, however, cannot be created since their characteristics are determined by the internal and external condition of the kernel or a software component. Registering these objects to the kernel or a software component is referred to as defining objects.

In general, positive serial numbers starting from 1 are used as object IDs. When the objects are classified for protection mechanism reasons into user objects and system objects, increasing positive serial numbers starting from 1 are used for user object ID numbers, and decreasing negative serial numbers starting from (–5) are used for system object ID numbers. In this case, only user objects are subject to automatic ID assignment. ID numbers from (–4) to 0 are reserved for special purposes.

[Standard Profile]

The Standard Profile does not require object classification nor does it require support for negative ID numbers. At the very least, positive ID numbers from 1 to 255 must be supported.

[Supplemental Information]

Interrupt handlers and rendezvous are examples of objects identified by object numbers. Object numbers are assigned to interrupt handlers according to hardware requirements while for rendezvous, object numbers are assigned based on the kernel's internal requirements. For these two types of objects, the application cannot freely assign numbers.

2.1.4 Priorities

Priorities are parameters determined by applications to control the processing order of tasks, messages, and so on. Positive serial numbers starting from 1 are used to represent priorities, where a smaller number indicates a higher precedence.

[Standard Profile]

In the Standard Profile, the kernel must support at least 16 different levels of task priority (from 1 through 16). The number of message priority levels must be equal to or greater than the number of task priority levels.

[Differences from the µITRON3.0 Specification]

The µITRON3.0 Specification allowed negative numbers to be used for system priorities; however, since negative values were seldom used, system priorities are limited to positive numbers in the µITRON4.0 Specification. Negative priorities are allowed but they are implementation-specific. µITRON3.0 requires at least 8 priority levels (1–8).

While the µITRON4.0 Specification does not specify the minimum number of priority levels, the Standard Profile requires it to support at least 16 priority levels (1–16).

2.1.5 Function Codes

Function codes are numbers assigned to identify service calls. Invoking a service call from a software interrupt, for instance, makes use of a function code. However function codes are not necessary in invoking a service call from a subroutine.

In the ITRON Specifications, each service call of a kernel or a software component is assigned a unique negative number as a function code. However, (–4) to 0 are reserved for special purposes. Positive function codes represent extended service calls.

2.1.6 Return Values of Service Calls and Error Codes

In principle, the return value of a service call is a signed integer. If an error occurs during the execution of a service call, an error code with a negative value is returned. A service call returns E_OK (= 0) or a positive integer if it completes its execution normally. Each service call specifies the meaning of its return value during normal completion. However service calls returning boolean values (BOOL type) and service calls that never return are exceptions. A service call that never returns should be declared as a function without a return value (i.e. a void type function) in a C language API.

An error code is divided into two parts, the main error code represented by the lower 8 bits, and the sub error code represented by the remaining bits. Both the main error code and the sub error code are negative, where the value of the sub error code is the result of arithmetically shifting the error code to the right by 8 bits. The resulting combined error code is also negative. The names, meanings, and values of the main error codes, defined under the ITRON General Definitions section, are common among the kernel and software components. Main error codes are classified into error classes, according to the situations in which they occur and also according to the need for error detection.

In the functional descriptions of service calls in the ITRON Specifications, only the main error codes returned by service calls are described, while sub error codes are implementation-defined. Sub error codes are also specified in some software component specifications. Descriptions, such as "an E_*XXXXX* error is returned" or "an E_*XXXXX* error occurs," included within the functional descriptions of service calls indicate that the service call returns an error code with a main error code of E_*XXXXX*.

In principle, unless the main error code is classified as a warning class error, side effects due to a service call that returns an error code do not arise. In other words, the invocation of a service call does not change the system state. However, service calls with unavoidable side effects are exceptions to the above principle. Side effects due to a service call must be explicitly specified in the service call's functional description.

The ITRON Specifications allows an implementation to omit detection of some errors in order to reduce kernel overhead. In principle, the main error code's class determines if the error detection can be omitted. Each error class explicitly mentions if the detection of its errors can be omitted. Exceptions to this principle are explicitly described in the service call's functional description. In the case where an error that should have been detected but was not because the error detection was omitted, the resulting system behavior is undefined.

The following main error codes occur in many, or almost all, of the service calls, thus they are not described in every service call.

E_SYS	System error
E_NOSPT	Unsupported function
E_RSFN	Reserved function code
E_CTX	Context error
E_MACV	Memory access violation
E_OACV	Object access violation
E_NOMEM	Insufficient memory

However, if these errors occur as a result unique to a service call, they are listed in the service call's description.

The error code returned by a service call that detects multiple errors is implementation-dependent.

[Supplemental Information]

The return value of E_OK (= 0) represents normal completion and is not an error code. However, for convince reasons, there are cases where it is described as an error code returned from a service call.

It is insufficient to simply examine the lower 8 bits of a return value for a negative number to determine whether the service call returned an error or not. This is because the lower 8 bits can be negative even when the service call completes normally and returns a positive value.

[Differences from the µITRON3.0 Specification]

In the µITRON4.0 Specification, an error code now consists of two parts, the main error code and the sub error code. Main error codes are shared between the kernel and software components. Sub error codes are intended to report the detailed cause of errors, and to be used mainly for debugging purposes. For example, when the main error code is E_PAR (parameter error), the sub error code can be used to indicate which parameter has an incorrect value. E_OK is not regarded as an error code.

Omitting error detection is explicitly permitted depending on the error class. Error codes which are not listed in each service call have been revised.

The μITRON3.0 Specification assumed the case where the return value of a service call is positive even though there were no service calls with a positive returned value. In the μITRON4.0 Specification, however, kernel service calls with positive return values exist. Also service calls that return boolean values have been introduced.

2.1.7 Object Attributes and Extended Information

Objects identified with ID numbers have object attributes while objects identified with object numbers, on the other hand, may or may not have object attributes. Object attributes that determine the operational mode and initial state of an object are defined when an object is registered. An object with an attribute value TA_*XXXXX* is called “an object with the TA_*XXXXX* attribute.” There is no interface available to read the object attributes after the object is registered.

The values and meanings of available object attributes are defined in the functional descriptions of the service calls or static APIs that register the objects. TA_NULL (= 0) is used when there is no need to specify the object attribute.

A processing unit object may have extended information. The extended information is specified at registration and is passed as a parameter when the object starts to execute. Extended information does not have any effects on the operation of the kernel or a software component. There is no interface available to read the extended information from a specific object.

[Supplemental Information]

Examples of processing unit objects with extended information are tasks, interrupt service routines, and time event handlers such as cyclic handlers.

[Differences from the μITRON3.0 Specification]

In the μITRON3.0 Specification, objects identified with ID numbers must have extended information, whereas in the μITRON4.0 Specification extended information is only provided when necessary. Extended information is now passed as a parameter when the object starts to execute and it cannot be read by object state reference service calls.

2.1.8 Timeout and Non-Blocking

Timeout or non-blocking features, when necessary, can be made available to service calls that might enter the WAITING state.

When a service call’s process is not completed within a specified time, the timeout feature cancels any further processing and returns from the service call immediately. In this case, the service call returns an E_TMOUT error. Since there are no side effects due to service calls returning an error, the system state, upon returning from the timed-out service call remains unchanged. However, some service calls due to their natures might prevent the system from proper restoration after the timeout cancellation. These exceptional cases should be explicitly specified in the service call's functional description.

When the timeout duration of a service call is set to 0, the service call does not enter the WAITING state even though it should. Setting the timeout duration of a service call to 0 is called polling. Service calls that execute polling never enter the WAITING state. The polling feature differs from the non-blocking feature described below in that polling cancels processing of the service call while non-blocking continues processing the service call.

In the non-blocking feature, a service call that enters the WAITING state returns immediately with an E_WBLK error but the processing still continues. The application program is notified by some means when the process completes or when it is canceled. Since the service call continues operating even after returning from its call, packets and data areas used for parameters and return parameters should not be used for other purposes until the process completes.

Processing of a service call is referred to as "pending" when it is in the WAITING state within the service call or when it continues operation due to a non-blocking service call.

The functional descriptions of the service calls in the ITRON Specifications describes the behavior when the service calls have no timeout, that is the behavior when the service calls wait forever. The description "entering the WAITING state" or "moved to the WAITING state" in the functional descriptions of the service calls do not imply any specific waiting duration. When a service call is invoked with a timeout duration, the service call returns with E_TMOUT as the main error code when the duration expires. In the case of polling, the service call does not enter the WAITING state and returns immediately with E_TMOUT as the main error code. With the non-blocking feature, the service call does not enter the WAITING state and returns E_WBLK as the main error code.

When specifying the timeout duration, TMO type, a positive value specifies the length of the timeout duration, TMO_POL (= 0) specifies polling, and TMO_FEVR (= –1) specifies the timeout duration should be forever. TMO_NBLK (= –2) can also be specified to indicate the non-blocking feature, depending on the service call. When the timeout duration is specified, it must be guaranteed that the timeout action occurs after at least the timeout duration has elapsed from the time the service call is invoked.

[Supplemental Information]

Kernel service calls do not have the non-blocking feature. Since a service call that executes polling never enters the WAITING state, the precedence of the invoking task remains unchanged.

In typical implementations, if the timeout duration is set to 1, the timeout action will occur at the second time tick after the service call is invoked. Since the timeout duration cannot be set to 0 (because 0 is assigned to TMO_POL), the system never times out on the first time tick after the service call is invoked.

2.1.9 Relative Time and System Time

Relative time of RELTIM type is used when specifying the time for an event to occur with respect to a certain time such as the time when a service call is invoked. When relative time is used, it must be guaranteed that the event occurs after at least the specified duration time elapsed.

Relative time can also be used to specify time-related actions other than event times, such as time intervals between events, where the meaning of relative time is define for each case.

System time of SYSTIM type is used when specifying absolute time. A function to set the current system time is available in the kernel specification. Changing the system time using this kernel function will not change the time in the real world (called real time) when an event specified using relative time is to occur. However, the system time when an event occurs will change.

[Supplemental Information]

In typical implementations, if the relative time is set to 1, the event will take place on the second time tick after the service call is invoked. If the relative time is set to 0, the event will take place on the first time tick after the service call is invoked.

2.1.10 System Configuration File

A system configuration file defines the configuration of the kernel and software components as well as the initial state of objects. It can contain static APIs for the kernel and software components, ITRON general static APIs (called general static APIs hereafter) and also C language preprocessor directives. A tool that interprets static APIs in a system configuration file and configures the kernel or a software component is called a configurator.

The steps in processing a system configuration file is as follows (see Figure 2-1). The system configuration file is first passed to the C language preprocessor. Then, it is passed on to each of the software component configurators and then, finally to the kernel configurator.

The software component configurator interprets the static APIs pertaining to itself and other general static APIs included in the file passed from the C preprocessor or from other previous configurators. The configurator then generates a source file, written in C language, that is necessary for configuring and initializing the software component itself. The software component configurator then adds static APIs for the next configurators when needed and removes the static APIs pertaining to itself to and from the passed files, before passing it on to the next configurator.

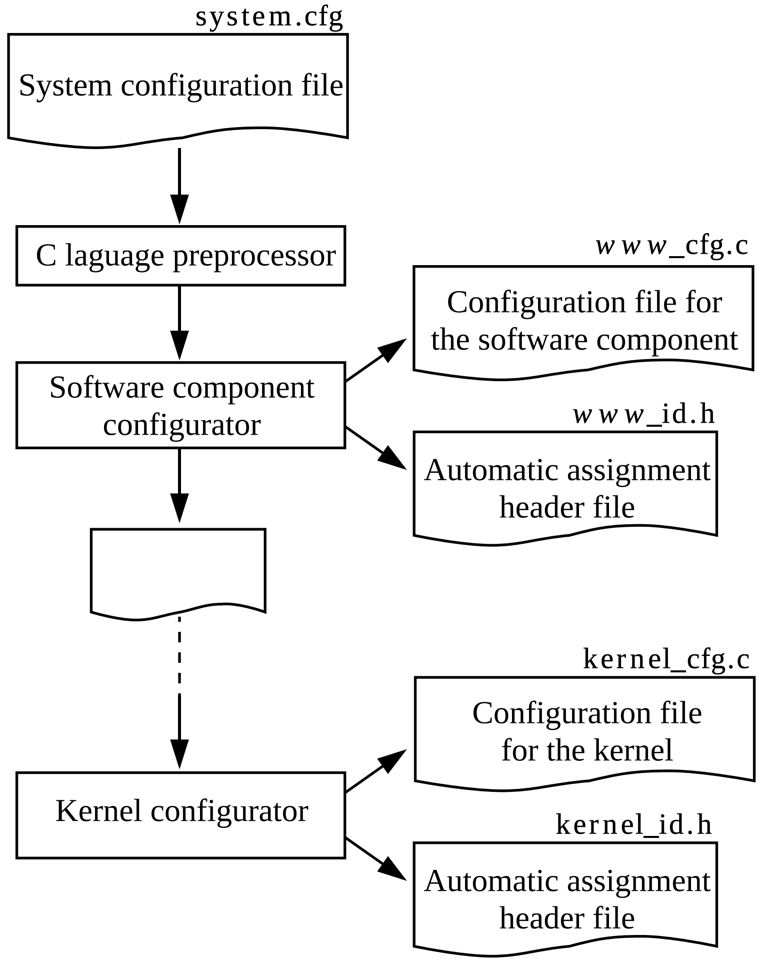


Figure 2-1. Processing Procedure for a System Configuration File

The kernel configurator interprets all static APIs included in the passed file and then generates a C language source file required for configuring and initializing the kernel. If it detects statements that cannot be interpreted either as a static API for the kernel or a general static API, the kernel configurator reports an error.

Kernel and software component configurators ignore any lines starting with a “#” sign. Software component configurators pass any lines starting with a “#” unchanged on to the next configurator.

[Supplemental Information]

Static APIs added by a software component configurator for the next configurators should not use preprocessor macros that are defined in the system configuration file and other files included through the preprocessor directive “#include.” The reason is that these preprocessor macros are already expanded after going through the C language preprocessor.

[Rationale]

The steps in processing a system configuration file is standardized to effectively deal with cases where the kernel and software components are developed independently.

Passing the system configuration file first to the C language preprocessor makes the following things possible.

- It allows a system configuration file to be divided into multiple files through the use of the "#include" directive. For example, when embedding a software component into a system, the necessary static APIs can be written in independent files. Those files can then be included in the system configuration file.
- It allows macros to be used instead of raw integers to define object ID numbers and object numbers.
- It allows conditional inclusion of configuration statements through the use of directives such as "#ifdef." In turn, it makes the changing of kernel and software component configurations and the initial states of objects possible.

Configurators ignore lines starting with "#" because these lines usually pertain to information generated by the preprocessor regarding the source file. However, configurators can still use these lines for informational purposes, such as generating error messages.

2.1.11 Syntax and Parameters of Static APIs

The syntax of the static APIs is based on the syntax of the C language function call. The parameters of a static API is based on the parameters of the corresponding service call in the C language API. However, if a parameter is a pointer to a packet, the elements of the packet should be separated with a comma "," and enclosed with braces "{" and "}."

The static API parameters are classified into the following four classes, according to available expressions:

(a) Integer Parameters with Automatic Assignment

A parameters of this class can be an integer (including a negative integer), an identifier, or a preprocessor macro (other than the restrictions mentioned below) which expands to either an integer or an identifier. Example parameters of this class are object ID numbers that are automatically assigned.

When a parameter of this class takes on the form of an identifier, the configurator responsible for processing the static API containing that identifier assigns an integer to that identifier. This assignment is called automatic ID number assignment by a configurator. The configurator generates a header file containing the macro definitions assigning integers to each identifier. Once the configurator assigns an integer to an identifier, the identifier can be used in the same manner as a preprocessor macro which expands to the assigned integer within static APIs that are processed by the configurator itself and by the following configurators.

(b) Integer Parameters without Automatic Assignment

A parameter of this class can only be an integer (including a negative integer) or a preprocessor macro (other than the restrictions mentioned below) which expands to an integer. Example parameters of this class are object ID numbers that cannot be automatically assigned and object numbers.

(c) Preprocessor Constant Expression Parameters

A parameter of this class is a constant expression that can be interpreted by a preprocessor. Only constants, macros, and operators that can be interpreted by a preprocessor can be used. Example parameters of this class are object attributes.

(d) General Constant Expression Parameters

A parameter of this class is any constant expression allowable in the C language. Most parameters belong to this class.

Each static API defines the class of its parameters. Integer parameters with or without automatic assignment and preprocessor constant expression parameters should be explicitly mentioned in the functional descriptions of static APIs. Other parameters not mentioned are assumed to be general constant expression parameters.

An ITRON general static API exists to include a file. Thus, there are two methods of including a file in a system configuration file: using the preprocessor directive "#include" or using the general static API. The differences between these two methods are described below:

- If preprocessor macros are used to define integer parameters with or without automatic assignment (hereafter, simply called integer parameters), only preprocessor macros defined in the system configuration file or other files included through a preprocessor directive can be used.
- Files included using preprocessor directives can contain only static APIs and preprocessor directives. In contrast, files included using general static APIs can only contain preprocessor directives and declarations and definitions in the C language.

NULL, which is often used to indicate that the kernel must allocate a memory area, is recognized as a symbol for static API parameters. A constant expression with a value 0 is not always interpreted as NULL. The behavior of such constant expression is implementation-dependent. Therefore, a NULL must not be macro-expanded by a preprocessor before a configurator processes it. In other words, NULL should not be defined as a preprocessor macro in a system configuration file or other files included through preprocessor directives.

The configurator reports errors when it detects syntax errors or incorrect number of parameters in static APIs. The method of handling errors found during the processing of static APIs is implementation-defined.

[Standard Profile]

In most static APIs, implementation-specific parameters can be added. In order for such implementations to conform to the Standard Profile, the configurator must correctly process the static APIs even when no implementation-specific parameters appear in the system configuration file. One of the methods to realize this is by supplying default values for implementation-specific parameters.

[Supplemental Information]

Static APIs can be written in free format inside a system configuration file. There may be white spaces, new lines and comments between words. The semicolon ";" is required at the end of each static API statement.

Since C language enumerated constants and "sizeof" cannot be interpreted by a preprocessor, they cannot be used in preprocessor constant expression parameters.

Removing a NULL preprocessor macro definition from a file that is included into the system configuration file through a preprocessor directive is sometimes difficult because of the file's structure. This problem can be solved in the following way. Define a specific identifier (for example, "CONFIGURATOR") as a preprocessor macro ("#define CONFIGURATOR") at the top of the system configuration file. Then, enclose the NULL preprocessor macro definition within "#ifndef CONFIGURATOR" and "#endif" directives.

[Rationale]

In order to simplify configurator implementations, static API parameters are classified into four classes. A configurator must be able to determine object ID numbers and object numbers properly so, excluding those that can be automatically assigned, object ID numbers and object numbers are limited to those expanded to integers after preprocessing (integer value parameter). Some parameters, such as object attributes, may have an effect on a registered object's structure depending on its value. To be able to use conditional directives based on these parameters in a C source file generated by a configurator, only expressions whose values can be determined by the preprocessor are allowed (preprocessor constant expression parameters). For other parameters, any constant expression in the C language is allowed (general constant expression parameters). If a configurator is implemented in this manner, it would not be able to determine all parameter values. Thus, its error checking capability is limited. Determining all the parameter values are possible by calling a compiler from the configurator and converting the constant expressions to values. However, since this approach requires modifying the configurator for each compiler, it has not been adopted as the standard method.

2.2 API Naming Convention

2.2.1 Software Component Identifiers

Software component identifiers are used to distinguish one set of standardized software component APIs from another. The software component identifier is made up of two to four characters. If a software component contains more than one functional unit, each individual unit may have a software component identifier. Software component identifiers are defined in the software component specification.

Software components that define their own APIs are not subject to this convention. However, to avoid naming conflicts with standardized software components, making the software component identifiers 5 or more characters long, or prefixing the identifier with "v" is recommended.

Hereafter, software component identifiers in lowercase are described as *www*, and those in uppercase as *WWW*.

2.2.2 Service Calls

The standard form of a kernel service call name takes the form of *xxx_yyy*, where *xxx* represents an operational procedure and *yyy* represents the target object of the operation. A service call derived from an *xxx_yyy* service call should be prefixed with the letter *z* resulting in a name of *zxxx_yyy*. If a service call is derived from a previously derived service call *zxxx_yyy*, the name becomes *zzxxx_yyy*.

Names of service calls for software components take the form of *www_xxx_yyy* or *www_zxxx_yyy*.

For naming implementation-specific service calls, the convention is to prefix "v" before *xxx* or *zxxx*. This creates standard names of the form v*xxx_yyy*, v*zxxx_yyy*, *www*_v*xxx_yyy*, or *www*_v*zxxx_yyy*. However, in the kernel specification, when a service call begins with an "i", which indicates that the service call can be invoked from interrupt handlers, the service call's name takes the form iv*xxx_yyy* instead of vi*xxx_yyy*.

[Supplemental Information]

Table 2-1 shows the abbreviations of the form *xxx*, *yyy*, and *z* used in the µITRON4.0 Specification and their English origin.

2.2.3 Callbacks

Since callback names are used as parameters, the naming convention for callbacks is the same as that of parameters.

2.2.4 Static APIs

Generally, static APIs are named by capitalizing all the letters of the corresponding service call names. The names of static APIs that have no corresponding service call follow the naming convention of service calls, with the names still capitalized.

The names and meanings of ITRON general static APIs that are used both by the kernel and software components are specified in the ITRON General Definitions section.

Table 2-1. Abbreviations used in the μITRON4.0 Specification and English origin

xxx	English origin
acp	accept
act*	activate
att	attach
cal	call
can	cancel
chg	change
clr	clear
cre	create
def	define
del	delete
dis	disable
dly	delay
ena	enable
exd	exit and delete
ext	exit
fwd	forward
get	get
loc*	lock
pol	poll
ras	raise
rcv	receive
ref	refer
rel	release
rot	rotate
rpl	reply
rsm	resume
set	set
sig	signal
slp	sleep
snd	send
sns	sense
sta	start
stp	stop
sus	suspend
ter	terminate
unl	unlock
wai*	wait
wup*	wake up

yyy	English origin
alm	alarm handler
cfg	configuration
cpu	CPU
ctx	context
cyc	cyclic handler
dpn	dispatch pending
dsp	dispatch
dtq	data queue
exc	exception
flg	eventflag
inh	interrupt handler
ini	initialization
int	interrupt
isr	interrupt service routine
mbf	message buffer
mbx	mail box
mpf	fixed-sized memory pool
mpl	memory pool
mtx	mutex
ovr	overrun handler
por	port
pri	priority
rdq	ready queue
rdv	rendezvous
sem	semaphore
sys	system
svc	service call
tex	task exception
tid	task ID
tim	time
tsk	task
tst	task status
ver	version

z	English origin
a	auto ID assign
f	force
i	interrupt
p	poll
t	timeout

* Abbreviations with asterisks (*) are also used as a *yyy* abbreviation.

2.2.5 Parameter and Return Parameter

The names of parameters and return parameters are all lowercase and are four to seven characters in length. The following conventions apply to parameter and return parameter names:

–id	– ID (object ID number, ID type)
–no	– number (object number)
–atr	– attribute (object attribute, ATR type)
–stat	– state (object state, STAT type)
–mode	– mode (service call operational mode, MODE type)
–pri	– priority (priority, PRI type)
–sz	– size (in bytes, SIZE type or UINT type)
–cnt	– count (in units, UINT type)
–ptn	– pattern
–tim	– time
–cd	– code
i–	initial value of –
max–	maximum –
min–	minimum –
left–	quantity left of –
p_–	pointer to the memory area of a return parameter (or a parameter)
pk_–	pointer to a packet
pk_c*yyy*	pointer to a packet passed to cre_*yyy*
pk_d*yyy*	pointer to a packet passed to def_*yyy*
pk_r*yyy*	pointer to a packet passed to ref_*yyy*
pk_*www*_c*yyy*	pointer to a packet passed to *www*_cre_*yyy*
pk_*www*_d*yyy*	pointer to a packet passed to *www*_def_*yyy*
pk_*www*_r*yyy*	pointer to a packet passed to *www*_ref_*yyy*
ppk_–	pointer to the memory area of a pointer to a packet

If the names of the parameters and return parameters are identical, they are generally the same data type.

2.2.6 Data Types

The names of data types are all uppercase and are two to ten characters in length. The following conventions apply to data type names:

–P	Pointer data type
T_–	Packet (data structure) type
T_C*YYY*	Packet type passed to cre_*yyy*
T_R*YYY*	Packet type passed to ref_*yyy*

T_*WWW*_–	Data structure used by software components
T_*WWW*_C*YYY*	Packet type passed to *www*_cre_*yyy*
T_*WWW*_R*YYY*	Packet type passed to *www*_ref_*yyy*

The names and meanings of ITRON general data types that are used by both the kernel and software components are specified in the ITRON General Definitions section.

2.2.7 Constants

The names of constants are all uppercase and follow the convention described below.

(A) ITRON General Constants

The names of ITRON general constants that are used both by the kernel and software components have no particular naming convention. The names and their respective meanings and values are specified in the ITRON General Definitions section.

(B) Error Codes

Main error codes defined in the ITRON Specifications take the form E_*XXXXX*, where *XXXXX* is approximately two to five characters in length. The form EV_*XXXXX* is used for implementation-specific main error codes.

Sub error codes have no particular naming convention.

Error classes take the form EC_*XXXXX*, where *XXXXX* is approximately two to five characters.

(C) Other Constants

Other constants take the form T*UU*_*XXXXX* or T*UU*_*WWW*_*XXXXX*, where *UU* is approximately one to three characters in length, and *XXXXX* is approximately two to seven characters in length. Constants used for the same type of parameters or return parameters should have the same identifier *UU*. *TUU* can be omitted for software component constants that are frequently used in many service calls and callbacks. In this case, such constants take the form *WWW*_*XXXXX*.

In addition to the above conventions, the following conventions apply to other constant names:

TA_–	Object attribute
TFN_–	Service call function code
TFN_*XXX*_*YYY*	Function code of *xxx*_*yyy*
TFN_*WWW*_*XXX*_*YYY*	Function code of *www*_*xxx*_*yyy*
TSZ_–	size of –
TBIT_–	bit size of –
TMAX_–	maximum –
TMIN_–	minimum –

2.2.8 Macros

The names of macros are all uppercase and conform to the naming convention for constants. The names and meanings of ITRON general macros that are used by both the kernel and software components are specified in the ITRON General Definitions section.

2.2.9 Header Files

The header file containing the definitions of data types, constants and macros, and other definitions specified in ITRON General Definitions section is named "itron.h." The header file containing all the service call declarations, data types, constants, and macro definitions specified in the kernel specification are named "kernel.h." The automatic assignment header file generated by the kernel configurator is named "kernel_id.h."

Header files containing service call declarations and other definitions specified in a software component specification are generally named beginning with the software component identifier. The automatic assignment header file generated by the software component configurator is named in a similar manner. The names of these header files are specified in the software component specification.

2.2.10 Kernel and Software Component Internal Identifiers

Internal identifiers are symbols registered to an object file's symbol table for external access. They are used within the kernel or a software component usually to refer to routines and memory areas. Kernel and software component internal identifiers should adhere to the naming convention defined below to avoid conflicts with other identifiers of an application program.

The names of kernel internal identifiers should begin with _kernel_ or _KERNEL_ at the C language level. The names of software component internal identifiers should begin with _*www*_ or _*WWW*_ at the C language level.

2.3 ITRON General Definitions

2.3.1 ITRON General Data Types

The ITRON general data types are as follows:

B	Signed 8-bit integer
H	Signed 16-bit integer
W	Signed 32-bit integer
D	Signed 64-bit integer

UB	Unsigned 8-bit integer
UH	Unsigned 16-bit integer
UW	Unsigned 32-bit integer
UD	Unsigned 64-bit integer
VB	8-bit value with unknown data type
VH	16-bit value with unknown data type
VW	32-bit value with unknown data type
VD	64-bit value with unknown data type
VP	Pointer to an unknown data type
FP	Processing unit start address (pointer to a function)
INT	Signed integer for the processor
UINT	Unsigned integer for the processor
BOOL	Boolean value (TRUE or FALSE)
FN	Function code (signed integer)
ER	Error code (signed integer)
ID	Object ID number (signed integer)
ATR	Object attribute (unsigned integer)
STAT	Object state (unsigned integer)
MODE	Service call operational mode (unsigned integer)
PRI	Priority (signed integer)
SIZE	Memory area size (unsigned integer)
TMO	Timeout (signed integer, unit of time is implementation-defined)
RELTIM	Relative time (unsigned integer, unit of time is implementation-defined)
SYSTIM	System time (unsigned integer, unit of time is implementation-defined)
VP_INT	Pointer to an unknown data type, or a signed integer for the processor
ER_BOOL	Error code or a boolean value (signed integer)
ER_ID	Error code or an object ID number (signed integers and negative ID numbers cannot be represented)
ER_UINT	Error code or an unsigned integer (the number of available bits for an unsigned integer is one bit shorter than UINT)

VB, VH, VW, VD, and VP_INT types are implementation-defined. Explicit type cast is necessary during access or assignment of values to variables of these data types.

In the case where the number of bits needed to represent the system time exceeds the number of bits of an integer, SYSTIM can be defined as a data structure where the structure's contents are implementation-defined.

[Standard Profile]

In the Standard Profile, 64-bit integer data types (D, UD, and VD) included in the ITRON general data types need not be supported.

In addition, the Standard Profile defines the minimum number of bits and the unit of time of the ITRON general data types as follows:

INT	16 or more bits
UINT	16 or more bits
FN	16 or more bits
ER	8 or more bits
ID	16 or more bits
ATR	8 or more bits
STAT	16 or more bits
MODE	8 or more bits
PRI	16 or more bits
SIZE	equal to the number of bits in a pointer
TMO	16 or more bits, unit of time is 1 msec
RELTIM	16 or more bits, unit of time is 1 msec
SYSTIM	16 or more bits, unit of time is 1 msec

[Supplemental Information]

SIZE is used to refer to the size of a large memory area, such as the stack size of a task or an entire variable memory pool size. UINT is used to refer to the size of a smaller memory area like a message length.

When SYSTIM is defined as a structure, variables of SYSTIM type cannot be manipulated by operators such as "+" and "–." In order to maintain the portability of an application program even in this case, operations on SYSTIM variables should be done using C language function calls and an operation module compatible with the definition of SYSTIM should be made available for each implementation.

[Differences from the μITRON3.0 Specification]

CYCTIME, ALMTIME, and DLYTIME are replaced by RELTIM. SYSTIME has been renamed to SYSTIM. STAT, MODE, and SIZE have been added. Complex data types VP_INT, ER_BOOL, ER_ID, and ER_UINT have been added while BOOL_ID has been removed. The size of a memory area is now handled using unsigned integers.

2.3.2 ITRON General Constants

(1) General Constants

The ITRON general constants are as follows:

NULL	0	Invalid pointer
TRUE	1	True
FALSE	0	False
E_OK	0	Normal completion

[Differences from the μITRON3.0 Specification]

The invalid pointer has been changed from NADR (= –1) to NULL (= 0) for compatibility with the C language.

(2) Main Error Codes

There are ten classes of main error codes as defined below:

(A) Internal Error Class (EC_SYS, from –5 to –8)

This class represents internal errors occurring inside the kernel or a software component. Omission of error detection of this class is implementation-defined.

E_SYS –5 System error

This error code indicates an internal error of unknown cause occurred inside the kernel or a software component.

(B) Unsupported Error Class (EC_NOSPT, from –9 to –16)

This class represents errors due to functions that are either not specified in the ITRON Specifications or are not supported by the implementation. Omission of error detection of this class is implementation-defined.

E_NOSPT –9 Unsupported function

This error code indicates that the function is specified in the ITRON Specifications but is not supported by the implementation. This error is returned if a part of or all of the service call functionality is not supported. Errors falling under E_RSFN and E_RSATR are not covered by this error code.

E_RSFN –10 Reserved function code

This error code indicates that a specified function code is not supported either in the ITRON Specifications or by the implementation. This error occurs when a service call is invoked from a software interrupt.

E_RSATR –11 Reserved attribute

This error code indicates that an attribute value is not supported either in the

ITRON Specifications or by the implementation.

(C) Parameter Error Class (EC_PAR, from –17 to –24)

This class represents errors due to parameters assigned with incorrect values. These errors can usually be detected statically. Omission of error detection of this class is implementation-defined.

E_PAR –17 Parameter error

This error code indicates that a parameter has an incorrect value that is usually statically detected. Errors falling under E_ID are not covered by this error code.

E_ID –18 Invalid ID number

This error code indicates that an object ID number is invalid. This error only occurs for objects identified by an ID numbers.

(D) Invoking Context Error Class (EC_CTX, from –25 to –32)

This class represents errors due to invocation of service calls from incorrect contexts. Omission of error detection of this class is implementation-defined.

E_CTX –25 Context error

This error code indicates that the context in which the service call is invoked is incorrect. Errors falling under E_MACV, E_OACV or E_ILUSE are not covered by this error code.

E_MACV –26 Memory access violation

This error code indicates that the specified memory area cannot be accessed from the context where the service call is invoked. This error is also returned if the specified memory area does not exist.

E_OACV –27 Object access violation

This error code indicates that the specified object cannot be accessed from the context where the service call is invoked. When the objects are classified into user objects and system objects, this error is returned if a system object is accessed from a context where access to system objects is prohibited.

E_ILUSE –28 Illegal service call use

This error code indicates that the use of the service call is incorrect. Occurrence of this error depends on the context from which the service call is invoked or on the state of the target object.

(E) Insufficient Resource Error Class (EC_NOMEM, from –33 to –40)

This class represents errors due to insufficient resources needed to execute the service call. Detection of errors of this class cannot be omitted.

E_NOMEM –33 Insufficient memory

This error code indicates that the service call failed to dynamically allocate enough memory for a memory area.

E_NOID –34 No ID number available

This error code indicates that there is no ID number available for the target object. This error is returned by the service call creating an object with an automatically assigned ID number.

(F) Object State Error Class (EC_OBJ, from –41 to –48)

This class represents errors due to the service call failing to execute because of the state of the target object. Since the occurrence of these errors depends on the state of the target object, they do not necessarily occur every time the same service call is invoked. Thus, dynamically checking for these errors is necessary. Error detection of this class cannot be omitted.

E_OBJ –41 Object state error

This error code indicates that the service call cannot be executed due to the state of the target object. Errors falling under E_NOEXS and E_QOVR are not covered by this error code.

E_NOEXS –42 Non-existent object

This error code indicates that the service call is not able to access the target object because the object does not exist. Since this error is returned only when the specified object ID number is within a valid range, the object can be created by specifying the same ID number that caused the error.

E_QOVR –43 Queue overflow

This error code indicates that the maximum queue limit or nesting level has been exceeded.

(G) Waiting Released Error Class (EC_RLWAI, from –49 to –56)

This class represents errors due to a waiting task being released from the WAITING state before its release condition is met. Detection of errors of this class cannot be omitted.

E_RLWAI –49 Forced release from waiting

This error code indicates that the waiting task is forcibly released from waiting or that the waiting process is cancelled.

E_TMOUT –50 Polling failure or timeout

This error code indicates that the polling service call has failed or that the service call made with a timeout has expired.

E_DLT –51 Waiting object deleted

This error code indicates that the object the task is waiting for has been deleted.

E_CLS –52 Waiting object state changed

This error code indicates that the service call cannot be executed due to a change in the state of the object the service call is waiting for. When the state change happened before the service call is invoked, the invoking task immediately returns with this error without moving into the WAITING state.

[Supplemental Information]

An example of the E_CLS error usage is in a service call that receives data through a communication line. E_CLS can be used to indicate that the connection is abnormally disconnected while the service call is waiting to receive data. The same error code can also be used even when the abnormal disconnection occurred before the service call was invoked.

(H) Warning Class (EC_WARN, from –57 to –64)

This class represents errors indicating that there are warnings associated with the service call's execution. Errors in this class are exceptions to the general rule stating that there are no side effects on the system state when a service call returns an error. That is, execution of service calls returning errors of this class can cause side effects on the system state. Detection of errors of this class cannot be omitted.

E_WBLK –57 Non-blocking call accepted

This error code indicates that the non-blocking service call is currently being executed.

E_BOVR –58 Buffer overflow

This error code indicates that a part of the received data was discarded due to buffer overflow.

(I) Reserved Error Codes (from –5 to –96 except those defined above)

These main error codes are reserved for future versions of the ITRON Specifications.

(J) Implementation-Specific Error Codes (from –97 to –128)

These main error codes are used for implementation-specific errors. The names of these main error codes must be of the form EV_*XXXXX*.

[Differences from the μITRON3.0 Specification]

Main error codes E_ILUSE and E_NOID have been added for new functionalities of the kernel specification, and E_CLS, E_WBLK, and E_BOVR have been added for software component specifications. Connection function errors of the form EN_*XXXXX*, and E_INOSPT, which were exclusive to ITRON/FILE Specification, have been removed. Some of the main error codes were reclassified and their values reassigned. Because the main error code is in the lower 8-bits of the error code, the assigned value is designed so that its value as an 8-bit signed integer remains negative. The error number (errno) has been removed.

(3) Object Attribute

The ITRON general object attribute is:

TA_NULL	0	Object attribute unspecified

(4) Timeout Specification

The ITRON timeout specifications are as follows:

TMO_POL	0	Polling
TMO_FEVR	–1	Waiting forever
TMO_NBLK	–2	Non-blocking

2.3.3 ITRON General Macros

(1) Error Code Retrieving Macros

ER mercd = MERCD (ER ercd)

This macro retrieves the main error code from an error code.

ER sercd = SERCD (ER ercd)

This macro retrieves the sub error code from an error code.

2.3.4 ITRON General Static APIs

(1) File Inclusion

INCLUDE (string) ;

This static API includes the file containing preprocessor macro definitions, the C language declarations, and the definitions necessary to interpret preprocessor constant expressions and general constant expression parameters. The INCLUDE static API must be specified in a system configuration file. The parameter string must be of a form that can be placed after the preprocessor directive “#include” once the INCLUDE static API is processed.

[Supplemental Information]

Examples of file inclusion using the static API are as follows:

```
INCLUDE ( "<itron.h>" ) ;
INCLUDE ( "\"memory.h\"" ) ;
```

[Rationale]

The reason string parameters are used is to prevent the file name from being expanded by the preprocessor before the system configuration file is passed to the configurator.

Chapter 3 Concepts and Common Definitions in µITRON4.0

3.1 Glossary of Basic Terms

(1) Task and Invoking Task

The term “task” refers to a unit of concurrent processing. While program statements inside a single task are executed sequentially, statements of different tasks are executed concurrently. Multiple tasks are executed concurrently when seen from an application’s point of view. However, the tasks do not actually run in parallel but rather, they are executed one by one under the control of the kernel, using time-sharing techniques.

The task that invokes a service call is called the “invoking task.”

(2) Dispatching and Dispatcher

The act of switching the currently executing task on a processor with another, non-executing task is called “dispatching” (or “task dispatching”). The mechanism in the kernel that performs dispatching is called the “dispatcher” (or the “task dispatcher”).

(3) Scheduling and Scheduler

The process that determines which task is to be executed next is called “scheduling” (or “task scheduling”). The mechanism in the kernel that executes scheduling is called the “scheduler” (or the “task scheduler”). In typical implementations, the scheduler is included in service call routines and/or in the dispatcher.

(4) Context

The environment in which a program executes is generally called the program’s “context.” When two programs have the same context, then at least the processor mode and stack space should be the same. The term context, however, is from an application’s point of view and there can be tasks which execute in independent contexts but actually run in the same processor mode and the same stack space.

(5) Precedence

The criterion used to determine the order of program execution is called “precedence.” In principle, when a higher precedence program becomes executable, it will begin executing in place of the currently executing lower precedence program.

[Supplemental Information]

A "priority" is a parameter given by an application to control the order of task execution and the order of message delivery, while precedence is used to clarify the order of program execution in this specification. The precedence between tasks is determined by the task priorities.

3.2 Task States and Scheduling Rule

3.2.1 Task States

Task states are classified into five broad categories. The blocked state category can be further broken down into three sub-states. The RUNNING state and the READY state are both generically referred to as the runnable state.

(a) RUNNING state

When a task is in the RUNNING state, the task is currently executing. When non-task contexts, such as interrupt handlers, take over execution, the task that was executing remains in the RUNNING state unless otherwise specified.

(b) READY state

When a task is in the READY state, the task is ready to execute but it cannot, because a task with higher precedence is already executing. In other words, the task can execute at any time once its precedence becomes the highest among the tasks in the runnable state.

(c) Blocked state

When a task is in the blocked state, the task cannot execute because the conditions necessary for its execution have not yet been met. The task is waiting for specific conditions to be met before it can continue execution. When a task enters the blocked state, the task's execution environment including the program counter and registers are saved. When the task resumes executing from the blocked state, the program counter and registers are restored to their previous values. The blocked state can be further classified into three sub-states:

(c.1) WAITING state

When a task is in the WAITING state, the execution is blocked due to the invocation of a service call. The service call specifies the conditions that must be met before the task continues execution.

(c.2) SUSPENDED state

When a task is in the SUSPENDED state, the task has been forcibly made to halt execution by another task. However, the invoking task can also suspend itself in the μITRON4.0 Specification.

(c.3) WAITING-SUSPENDED state

When a task is in the WAITING-SUSPENDED state, the task is both waiting for a condition to be met and suspended. A task in the WAITING state will be moved to the WAITING-SUSPENDED state if there is a request to move it to the SUSPENDED state.

(d) DORMANT state

When a task is in the DORMANT state, the task is either not yet executing or has already finished. The context information of a task will not be saved while the task is in the DORMANT state. When a task is activated from the DORMANT state, it will begin executing from the task's start address. The contents of the registers when the task begins executing are not guaranteed unless otherwise specified.

(e) NON-EXISTENT state

This indicates a virtual state where the task in question does not exist in the system, either because it has not yet been created or because it has already been deleted.

There may be other transitional states, depending on the implementation, that cannot be classified into any states listed above. (see Section 3.5.6).

If a task which has been moved to the READY state has higher precedence than the task in the RUNNING state, the lower precedence task will be moved to the READY state and the higher precedence task will be dispatched and moved to the RUNNING state. In this case, we say that the task that was in the RUNNING state has been preempted by the task that was moved to the RUNNING state. Even if the functional description of a service call mentions that "a task is moved to the READY state," it may be moved directly to the RUNNING state depending on the task precedence.

Task activation means that a task in the DORMANT state is moved to the READY state. All states other than the DORMANT state and the NON-EXISTENT state are generically referred to as active states. Task termination means that a task in the active state is moved to the DORMANT state.

Releasing a task from waiting means that if the task is in the WAITING state, it will be moved to the READY state, and if the task is in the WAITING-SUSPENDED state, the task will be moved to the SUSPENDED state. Resuming a suspended task mean that if the task is in the SUSPENDED state, it will be moved to the READY state, and if the task is in the WAITING-SUSPENDED state, it will be moved to the WAITING state.

Figure 3-1 shows the task state transitions for typical implementations. There may be other state transitions, depending on the implementation, that are not shown in this figure.

[Supplemental Information]

The WAITING state and the SUSPENDED state are independent of each other. Therefore a request to move a task to the SUSPENDED state does not affect the release condition of the task. In other words, a waiting task's release condition does not change whether or not the task is in the WAITING state or in the WAITING-SUSPENDED

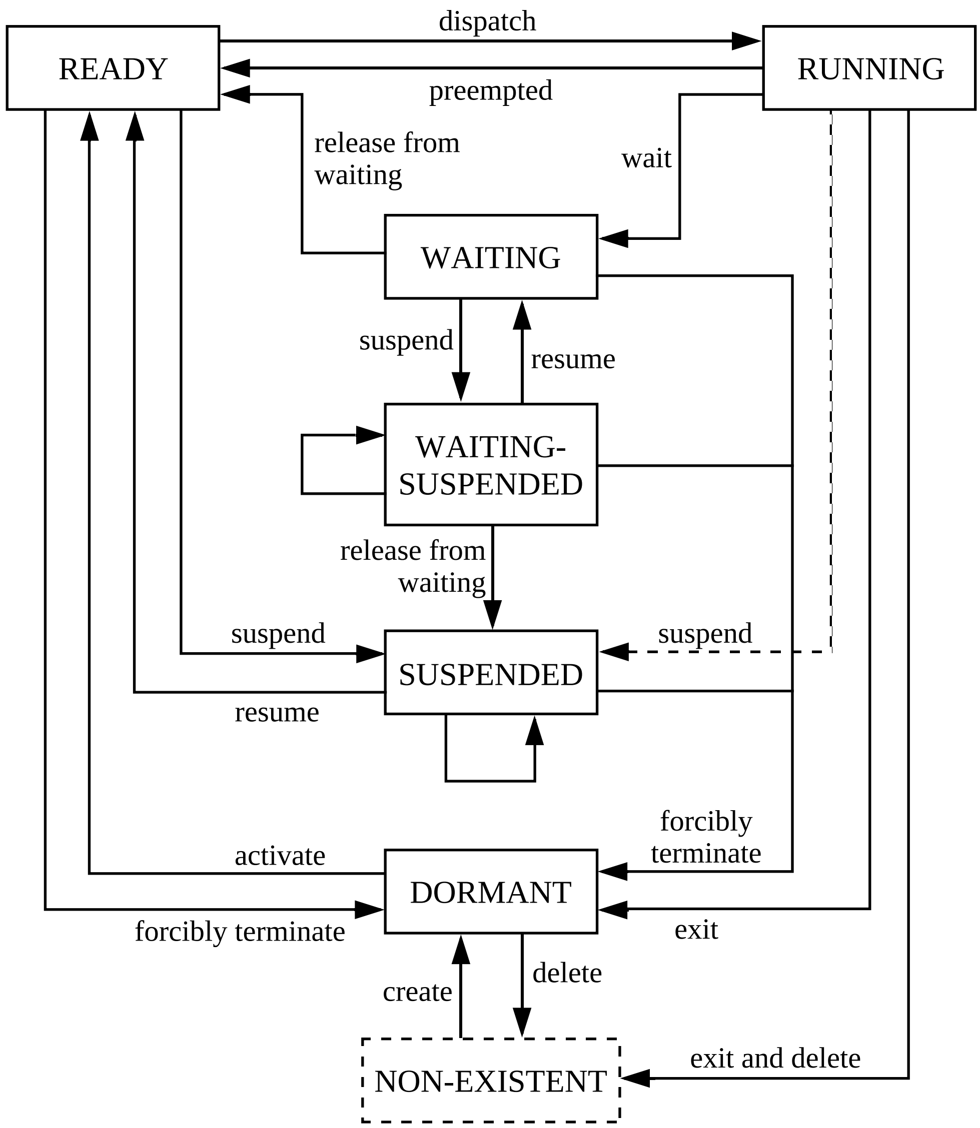


Figure 3-1. Task State Transitions

state. Therefore, if a task that is waiting for a resource (such as a semaphore resource or a memory block) is suspended and moved to the WAITING-SUSPENDED state, the task will still acquire the resource under the same conditions as it would in the WAITING state.

[Differences from the µITRON3.0 Specification]

The task state names are now in the adjective form. They have been renamed from RUN to RUNNING, from WAIT to WAITING, from SUSPEND to SUSPENDED, and from WAIT-SUSPEND to WAITING-SUSPENDED.

An invoking task can now move itself to the SUSPENDED state. This feature facilitates implementing APIs that do not distinguish self-suspension from suspension of other tasks (such as those for POSIX and Java threads) on µITRON4.0 Specification kernels

[Rationale]

The ITRON Specifications distinguishes the WAITING state from the SUSPENDED state because a task can exist in both states at the same time. Defining the overlapped state as the WAITING-SUSPENDED state makes the task state transition clearer and makes the understanding of service calls easier. Because tasks in the WAITING state cannot invoke service calls, they will never be in more than one kind of WAITING state, e.g. sleeping while waiting for a semaphore resource. In the ITRON Specifications, the SUSPENDED state is the only blocked state that can be caused by other tasks. Tasks may be suspended multiple times by other tasks. This is handled through nesting of the suspend requests.

3.2.2 Task Scheduling Rules

In the ITRON Specification, the preemptive, priority-based task scheduling is conducted based on the priorities assigned to tasks. If there are a number of tasks with the same priority, scheduling is conducted on a “first come, first served” (FCFS) basis. This task scheduling rule is defined using the precedence between tasks based on task priorities as described below.

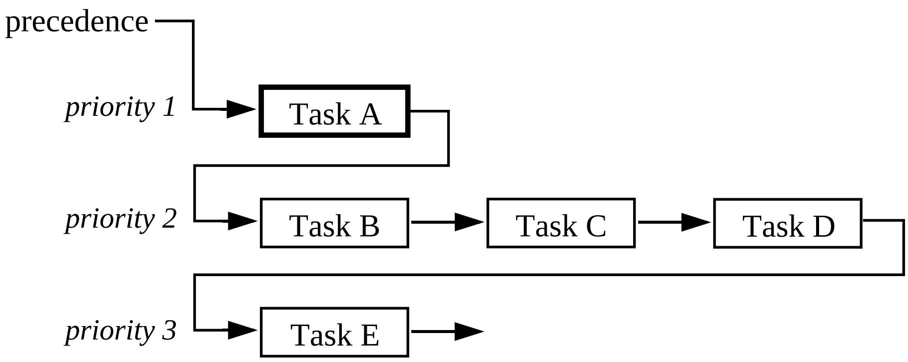
If more than one runnable task exists, the highest precedence task will be in the RUNNING state, and the rest in the READY state. Among the tasks with different priorities, the task with the higher priority has higher precedence. Among tasks of the same priority, the task that entered the runnable (RUNNING or READY) state earlier has higher precedence. However, the precedence between tasks of the same priority may change due to the invocation of some service calls.

When a task is given precedence over any other runnable tasks, a dispatch will occur immediately, and the task in the RUNNING state will be switched with the new task. However, when the system is in a state where dispatching does not occur, the switch of the task in the RUNNING state will wait until dispatching is allowed.

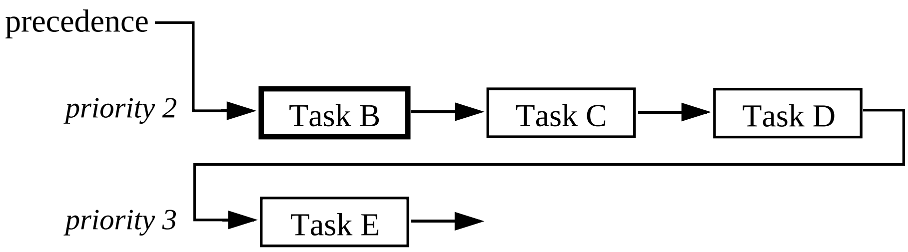
[Supplemental Information]

In the ITRON Specifications, as long as the highest precedence task is in the runnable state, no lower precedence tasks are allowed to execute. No other tasks will execute unless the highest precedence task cannot be executed for some reason, such as being placed in the WAITING state. In this respect, the scheduling rule of the ITRON Specifications differs entirely from TSS (Time-Sharing Systems), which attempts to execute multiple tasks as equally as possible. However, the precedence between tasks with the same priority may be modified through service calls. Applications can execute in a round-robin fashion, a common scheduling system for TSS, by using those service calls.

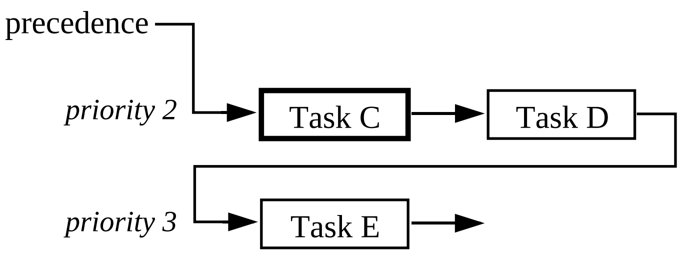
Figure 3-2 shows that among tasks of the same priority, the task that becomes runnable first has the highest precedence. Figure 3-2 (a) shows the precedence between tasks



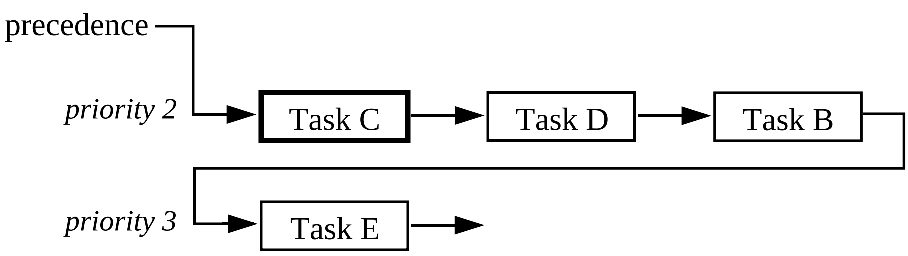
(a) Precedence in the first state



(b) Precedence after Task B becomes the RUNNING state



(c) Precedence after Task B becomes the blocked state



(d) Precedence after Task B is released from waiting

Figure 3-2. Precedence between Tasks

after Task A (priority 1), Task E (priority 3), and Task B, C and D (priority 2), have been activated in this order. Task A, with the highest precedence, is in the RUNNING state.

When Task A terminates, Task B, the task with the second highest precedence, moves to the RUNNING state (Figure 3-2 (b)). If Task A is reactivated, Task B will be preempted and return to the READY state. However, since Task B will be in the runnable state before Task C and Task D, it will have the highest precedence among the tasks with the same priority. This means that the priorities between tasks will go back to the state shown in Figure 3-2 (a).

When Task B changes from the runnable state to the WAITING state, the organization of the tasks will change from Figure 3-2 (b) to Figure 3-2 (c). If Task B is released from waiting, the priority of Task B will be the lowest among tasks of the same priority because Task B becomes runnable after Task C and Task D. This state is illustrated in Figure 3-2 (d).

To summarize, if a task in the READY state moves to the RUNNING state and then goes back to the READY state, it will have the highest precedence among tasks of the same priority. On the other hand, when a task in the RUNNING state moves to the WAITING state, and then back to the READY state, the task will have the lowest precedence among the tasks of the same priority.

[Differences from the µITRON3.0 Specification]

The ready queue is a concept related to the implementation, so in the specification "precedence" is used instead of "ready queue" to describe the scheduling rule.

To reduce implementation dependencies, a task that is moved from the SUSPENDED state to the READY state, will have the lowest precedence among the tasks of the same priority.

3.3 Interrupt Process Model

3.3.1 Interrupt Handlers and Interrupt Service Routines

In the µITRON4.0 Specification, interrupt handlers and interrupt service routines are processing units started by external interrupts (simply called as interrupts below).

Basically, execution of an interrupt handler depend on the processor architecture. Therefore, the interrupt handler, not the kernel, should be the one to control the Interrupt Request Controller (IRC). The implementation of an interrupt handler is implementation-defined because it generally depends on the processor interrupt architecture and the IRC. An interrupt handler cannot be ported as is to a different system.

An interrupt service routine is a routine started by an interrupt handler. It can be implemented independently from the processor architecture and the IRC used. This means that there is no need for the interrupt service routine to control the IRC since the interrupt handler starting the interrupt service routine already controls the IRC.

The µITRON4.0 Specification defines the APIs to register an interrupt handler prepared by the application, such as DEF_INH, and the APIs to register an interrupt service routine, such as ATT_ISR. An implementation should provide either one set of APIs or both sets. If the APIs for registering an interrupt handler are provided, the kernel can provide a glue routine for the interrupt handler that includes processes to be done before and after the interrupt handler executes. Depending on the interrupt handler attribute, the interrupt handler can be started through the provided glue routine. If only the APIs for registering an interrupt service routine are provided, the kernel must provide the interrupt handler that starts the interrupt service routine. Although both APIs are allowed at the same time, the behavior when both APIs are used is implementation-defined.

Depending on the implementation, the kernel does not control interrupts with higher priorities than a threshold priority level, including non-maskable interrupts. These kinds of interrupts are called non-kernel interrupts. The method for defining the threshold priority level is implementation-defined. No kernel service calls can be invoked from interrupt handlers started by non-kernel interrupts. In this specification document, the term "interrupt" and "interrupt handler" do not include non-kernel interrupts and interrupt handlers started by non-kernel interrupts, respectively.

Figure 3-3 shows the interrupt processing model in the μITRON4.0 Specification. This figure only outlines a conceptual model. The actual method used to realize interrupt processing depends on the application and implementation.

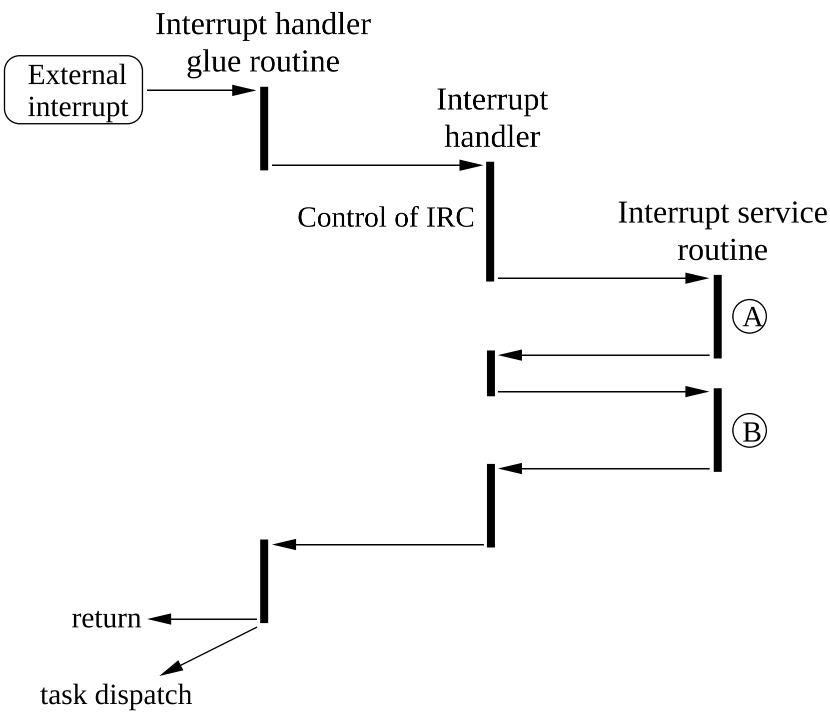


Figure 3-3. Interrupt Processing Model

[Supplemental Information]

The responsibilities of the interrupt handler glue routine include saving and restoring registers used within the handler, switching stack space, task dispatching, and returning from the interrupt. The operations actually performed by the glue routine depend on the implementation. The operations that are included in the glue routine and those that are included in the interrupt handler prepared by the application are implementation-defined and determined by the interrupt handler attributes.

The responsibilities of the interrupt handler that starts interrupt service routines include reading the cause of the interrupt from the IRC, branching based on the cause, clearing the edge trigger, and clearing the in-service flag of the IRC. In addition, the CPU must be unlocked before starting an interrupt service routine.

In order to reduce the overhead associated with an interrupt service routine, the interrupt handler glue routine and the interrupt handler can be merged. Interrupt service routines can be directly embedded in-line within the interrupt handler.

[Standard Profile]

The Standard Profile requires support for either the APIs to register an interrupt handler or the APIs to register an interrupt service routine.

[Rationale]

Interrupt service routines are introduced to improve the portability of an application's interrupt processing. Interrupt handlers, which are less portable, remain so that a kernel can be provided that is independent of an IRC.

3.3.2 Ways to Designate an Interrupt

In the µITRON4.0 Specification, there are two ways to designate an interrupt: by using an interrupt number and by using an interrupt handler number. In addition, an interrupt service routine is identified by an ID number.

The interrupt handler number, INHNO type, is used to designate the interrupt that is handled by an interrupt handler registered with the kernel. The designated interrupt should be able to be determined without referencing the IRC. The interrupt handler number corresponds to the interrupt vector number of the processor in typical implementations. When the processor does not have interrupt vectors, there may be only one available interrupt handler number.

The interrupt number, INTNO type, is used to designate the interrupt that is handled by an interrupt service routine registered with the kernel. The interrupt number is also used as a parameter to some service calls, such as `dis_int` and `ena_int`, to disable and enable each interrupt individually. Because starting an interrupt service routine and individually disabling/enabling interrupts are executed by controlling the IRC, the interrupt number corresponds to the interrupt request line of the IRC.

An interrupt service routine is bound to a specific interrupt request line from a device. Since the interrupt request line to the IRC can be connected to more than one device, more than one interrupt service routine can be registered to a single interrupt number. If the interrupt designated by the interrupt number occurs, all interrupt service routines bound to the interrupt number will be called one by one. The order in which the interrupt service routines are called is implementation-dependent. Multiple interrupt service routines bound to a single interrupt number are distinguished by interrupt service routine ID numbers.

[Supplemental Information]

For the case when multiple devices are connected to a single interrupt request line to the IRC, the devices may supply an interrupt vector number used by the processor to determine the actual source of the interrupt. In this case, interrupt sources supplying different vector numbers can have different interrupt numbers.

3.4 Exception Process Model

3.4.1 Exception Processing Framework

The µITRON4.0 Specification defines the CPU exception handling and the task exception handling functions.

A CPU exception handler is started when the processor detects an exception. A CPU exception handler can be registered by the application for each kind of CPU exception. The kernel can provide a glue routine for the CPU exception handler that includes processes to be done before and after the CPU exception handler executes. Depending on the CPU exception handler attribute, the CPU exception handler can be started through the provided glue routine.

Because the CPU exception handlers are common to the whole system, the context and the state at the point when the CPU exception occurred can be probed by the CPU exception handler. When a CPU exception occurs within a task, the CPU exception handler can let the task's exception handling routine handle the exception if desired.

The task exception handling functions are used to stop the normal execution of the specified task and to start the task's exception handling routine. The task's exception handling routine is executed within the same context as the task. When returning from the task exception handling routine, the execution of the interrupted execution will continue. The application can register one task exception handling routine for each task. The task exception handling functions will be explained in Section 4.3.

[Standard Profile]

The CPU exception handling routine and the task exception handler must be supported in the Standard Profile.

3.4.2 Operations within a CPU Exception Handler

The implementation method of a CPU exception handler is implementation-defined, because it generally depends on the processor exception handling architecture and the kernel implementation. A CPU exception handler cannot be ported to a different system without changes.

The service calls that can be invoked in a CPU exception handler are implementation-defined. However, a CPU exception handler must be able to perform the operations described below. The method to perform these operations is implementation-defined.

A CPU exception handler must be able to:

(a) Read the context and system state when the CPU exception occurred. The kernel must provide a method to reference the system state information when the CPU exception occurred that would normally be obtained through sns_*yyy* service calls invoked just prior to the CPU exception.

(b) Read the task ID of the task in which the CPU exception occurred, if the exception occurred while a task was executing.

(c) Request task exception handling. This operation is equivalent to invoking ras_tex within the CPU exception handler.

If the exception occurs while the CPU is locked, it is not necessary to support (b) and (c).

3.5 Context and System State

3.5.1 Processing Units and Their Contexts

In the μITRON4.0 Specification, the kernel controls the execution of the following processing units:

- (a) Interrupt handlers
  - (a.1) Interrupt service routines
- (b) Time event handlers
- (c) CPU exception handlers
- (d) Extended service call routines
- (e) Tasks
  - (e.1) Task exception handling routines

Interrupt handlers and interrupt service routines execute in their own independent contexts. For the remainder of this section, the descriptions about interrupt handlers apply to interrupt service routines as well, unless a specific description about interrupt service routines is provided.

Time event handlers are started by a time trigger. There are three kinds of time event handlers: cyclic handlers, alarm handlers, and overrun handlers. Time event handlers execute in their own independent contexts. Cyclic handlers are explained in Section 4.7.2, alarm handlers are explained in Section 4.7.3, and overrun handlers are explained in Section 4.7.4.

A CPU exception handler executes in an independent context determined by the CPU exception and by the context in which the CPU exception occurred.

Extended service call routines are registered by the application and are started by invoking extended service calls. An extended service call routine executes in an independent context determined by the extended service call and by the context from which the extended service call is invoked. Extended service call routines are explained in Section 4.10.

Tasks execute in their own independent contexts. A task exception handling routine executes in the associated task's context. In the remainder of this section, the descriptions about tasks apply to task exception handling routines as well, unless a specific description about task exception handling routines is provided.

Kernel processes are not classified into the processing units mentioned above. The kernel processes include service call execution, the dispatcher, glue routines for interrupt handlers (or interrupt service routines), and glue routines for CPU exception handlers. The context in which the kernel processes execute is not specified because it does not affect the behavior of the application.

[Differences from the µITRON3.0 Specification]

The term "time event handler" is now used instead of "timer handler." The term "extended service call routine" is now used instead of "extended SVC handler."

3.5.2 Task Contexts and Non-Task Contexts

Contexts that can be regarded as a part of a task are generically called task contexts, while other contexts are generically called non-task contexts.

Contexts in which tasks execute are classified as task contexts. Contexts in which interrupt handlers and time event handlers execute are classified as non-task contexts. Contexts for CPU exception handlers and for extended service call routines depend on the contexts where they occur or where they are invoked. These contexts are defined below.

When CPU exceptions occur in task contexts, the CPU exception handlers can execute either in task contexts or in non-task contexts. In this case, the context in which a CPU exception handler executes is implementation-defined. When CPU exceptions occur in non-task contexts, the CPU exception handlers execute in non-task contexts.

When extended service calls are invoked from task contexts, the extended service routines execute in task contexts. When extended service calls are invoked from non-task contexts, the extended service routines execute in non-task contexts.

In the µITRON4.0 Specification, service calls that can be invoked in task contexts and service calls that can be invoked in non-task contexts are distinguished from each other. The invocation of service calls in non-task contexts is described in Section 3.6.

The service calls that can move the invoking task to the blocked state and the service calls where the invoking task are implicitly specified may not be invoked from non-task contexts. If such service calls are invoked, an E_CTX error is returned. Using the parameter TSK_SELF (= 0), which designates the invoking task as a parameter of the service call, is also prohibited from non-task contexts. If TSK_SELF is used from non-task contexts, an E_ID error is returned.

[Supplemental Information]

As mentioned in Section 3.5.3, dispatching does not occur during a CPU exception handler execution, because the precedence of the CPU exception handler is higher than the precedence of the dispatcher. Therefore, in implementations where the CPU exception handler executes within task contexts, the behavior of service calls that may move the task to the blocked state is undefined in this specification. If an error should be reported under these conditions, an E_CTX error is returned.

[Differences from the µITRON3.0 Specification]

The terms “task contexts” and “non-task contexts” are now used instead of “task portions” and “task-independent portions.” The term “transitional state” has been removed because the context in which the kernel is executed is not specified. In the µITRON4.0 Specification, the concept of quasi-task portions is undefined and is included in task contexts, because the processor mode is not specified.

3.5.3 Execution Precedence and Service Call Atomicity

In the µITRON4.0 Specification, the precedence for executing each processing unit and the dispatcher is specified as follows:

(1) Interrupt handlers, time event handlers, CPU exception handlers
(2) Dispatcher (one of the kernel processes)
(3) Tasks

The precedence of interrupt handlers is higher than the precedence of the dispatcher. The precedence between interrupt handlers and interrupt service routines is implementation-defined, depending on interrupt priorities.

The precedence of time event handlers is implementation-defined. However, time event handlers cannot have higher precedence than interrupt handlers invoking isig_tim, and it must be higher than the precedence of the dispatcher.

The precedence of CPU exception handlers is higher than the precedence of the processing unit where the CPU exception occurs and higher than the precedence of the dispatcher. The precedence of CPU exception handlers relative to the precedence of interrupt handlers and time event handlers is implementation-defined.

The precedence of extended service call routines is higher than the precedence of the processing unit that invokes the extended service calls and is lower than the precedence of any processing unit that has a higher precedence than the invoking processing unit.

The precedence of tasks is lower than the precedence of the dispatcher. The relative precedence of tasks is defined by the task scheduling rule.

Basically, kernel service calls are executed atomically and the state of ongoing service call processes is invisible. However, the implementation may choose to modify this behavior to improve system response. In this case, service call operation must still appear to be executed atomically as far as the application can determine using service calls. This behavior is called the service call atomicity guarantee. Service call atomicity may be difficult to guarantee while maintaining a high level of response with implementation-specific functions not covered in this specification. If this is so, then loosening the principle of service call atomicity is permitted.

When kernel service calls are executed atomically, their precedence is highest. When the atomicity is loosened as described above, the precedence of service call processes is implementation-dependent as long as their precedence is higher than the processing unit invoking the service calls.

Other kernel processes than service call processes such as the dispatcher, glue routines for interrupt and exception handler are treated similarly.

[Standard Profile]

The Standard Profile requires service calls that are part of the Standard Profile must be guaranteed to operate atomically.

[Supplemental Information]

Since the precedence of the dispatcher is lower than the precedence of interrupt handlers, dispatching does not occur until all activated interrupt handlers are processed. This was called the “delayed dispatching” rule. The same applies to time event handlers and CPU exception handlers.

3.5.4 CPU Locked State

The CPU state of the system is in either the locked or unlocked state. In the CPU locked state, interrupt handlers (except for those started by a non-kernel interrupt) and time event handlers are not started and dispatching does not occur. The CPU locked state can be considered as the state in which the precedence of the executing processing unit is highest. There might be a transitional state that is neither the CPU locked state nor the CPU unlocked state, depending on the implementation.

The transition to the CPU locked state is called “locking the CPU,” while the transition to the CPU unlocked state is called “unlocking the CPU.”

In the CPU locked state, the following service calls can be invoked:

loc_cpu / iloc_cpu	lock the CPU
unl_cpu / iunl_cpu	unlock the CPU
sns_ctx	reference contexts

sns_loc	reference CPU state
sns_dsp	reference dispatching state
sns_dpn	reference dispatch pending state
sns_tex	reference task exception handling state

where loc_cpu/iloc_cpu means that loc_cpu may be called from task contexts and iloc_cpu from non-task contexts (the same rule applies to unl_cpu/iunl_cpu). The behavior of other service calls invoked from a CPU locked state is undefined. When an error should be reported, an E_CTX error will be returned.

The CPU state is implementation-dependent after an interrupt handler starts (either in the CPU locked state, in the CPU unlocked state, or in a transitional state). However, it is implementation-defined how to enter the CPU unlocked state in interrupt handlers. It is also implementation-defined how to return correctly from interrupt handlers after the system has entered the CPU unlocked state. The behavior is undefined when interrupt handlers do not return according to the method specified by the implementation.

The system is in the CPU unlocked state after interrupt service routines and time event handlers start. When returning from these routines/handlers, the system must be in the CPU unlocked state. The behavior is undefined when returning from these routines/handlers in the CPU locked state.

The start of and the return from CPU exception handlers do not change the CPU state. In other words, after CPU exception handlers start, the system is in the CPU locked (unlocked) state when the CPU exception occurs in the CPU locked (unlocked) state. When the CPU state is changed in CPU exception handlers, it should be returned to the previous state before returning from the CPU exception handlers. The behavior is undefined when returning from CPU exception handlers without returning to the previous state.

The start of and the return from extended service call routines do not change the CPU state. In other words, after extended service call routines start, the system is in the CPU locked (unlocked) state when the extended service calls are invoked in the CPU locked (unlocked) state. After returning from the extended call routines, the CPU state remains the same as set by the routines.

After tasks start, the system is in the CPU unlocked state. When tasks exit, the system must be in the CPU unlocked state. The behavior is undefined when tasks exit while in the CPU locked state.

The start of and the return from task exception handling routines do not change the CPU state. However, it is not specified whether task exception handling routines are started in the CPU locked state. After returning from the task exception handling routines, the CPU state remains the same as set by the routines.

[Supplemental Information]

Interrupts are usually, but not always, allowed in the CPU unlocked state.

[Differences from the µITRON3.0 Specification]

The meaning of the CPU state has changed. In the µITRON3.0 Specification, the CPU locked state was considered the state where interrupts and task dispatching were disabled. However, in the µITRON4.0 Specification, the CPU locked state is treated conceptually as a state independent of interrupts and task dispatching. In the CPU locked state only a few service calls can be invoked.

3.5.5 Dispatching Disabled State

The dispatching state of the system is either disabled or enabled. Dispatching does not occur in the dispatching disabled state. The dispatching disabled state can be considered as the state in which the precedence of the executing processing unit is higher than that of the dispatcher. There might be a transitional state that is neither the dispatching disabled state nor the dispatching enabled state, depending on the implementation.

The transition to the dispatching disabled state is called “disabling dispatching,” while the transition to dispatching enabled state is called “enabling dispatching.”

In the dispatching disabled state, service calls that can be invoked from task contexts have the following restrictions. While in the dispatching disabled state, the behavior caused by invoking service calls that can move the invoking task to the blocked state is undefined, unless otherwise specified. When an error should be reported, an E_CTX error will be returned. On the other hand, service calls that can be invoked from non-task contexts do not have restrictions even in the dispatching disabled state.

The start of and the return from interrupt handlers, interrupt service routines, time event handlers, and CPU exception handlers do not change the dispatching state. In other words, after these handlers/routines start, the system is in the dispatching disabled (enabled) state when these handlers/routines start in the dispatching disabled (enabled) state. When the dispatching state is changed in these handlers/routines, it should be returned to the previous state before returning from these handlers/routines. The behavior is undefined when returning from these handlers/routines without returning to the previous state.

The start of and the return from the extended service call routines do not change the dispatching state. In other words, after the extended service call routines start, the system is in the dispatching disabled (enabled) state when the extended service call routines are invoked from the dispatching disabled (enabled) state. After returning from the extended call routines, the dispatching state remains the same as set by the routines.

After tasks start, the system is in the dispatching enabled state. When tasks exit, the system must be in the dispatching enabled state. The behavior is undefined when tasks exit in the dispatching disabled state.

The start of and the return from the task exception handling routines do not change the dispatching state. In other words, after task exception handling routines start, the system is in the dispatching disabled (enabled) state when the task exception handling routines start from the dispatching disabled (enabled) state. After returning from the task exception handling routines, the dispatching state remains the same as set by the routines.

The dispatching state is treated independent of the CPU state.

[Supplemental Information]

The restriction that behavior is undefined when service calls that can move the invoking task to the blocked state are invoked while in the dispatching disabled state applies to a service call as a whole, unless otherwise specified. For example, service calls for polling, e.g. pol_sem, can be invoked in the dispatching disabled state because there is no possibility that the invoking task will enter the WAITING state. On the other hand, the behavior of service calls that may cause a task to enter the WAITING state, e.g. twai_sem, is undefined even if they are invoked with TMO_POL (polling) in the timeout parameter.

There are no service calls that change the dispatching state in non-task contexts in the µITRON4.0 Specification. Therefore, it is impossible to change the dispatching state within interrupt handlers and time event handlers unless an implementation-specific extension is provided. The same rule applies to CPU exception handlers when they are executed in non-task contexts.

The dispatching state is treated independently from the CPU state. Therefore, for example, if the system is in the dispatching disabled state and the CPU state changes from the locked state to the unlocked state, the system remains in the dispatching disabled state. The dispatching state can still be sensed while the system is in the CPU locked state.

[Differences from the µITRON3.0 Specification]

The meaning of the dispatching disabled state has been changed. The dispatching state is defined as a state treated independently of the CPU state.

3.5.6 Task State during Dispatch Pending State

Dispatching does not occur during execution of processing units with higher precedence than that of the dispatcher, and while in the CPU locked state or in the dispatching disabled state. These three conditions are collectively called the dispatch pending state. The task states in the dispatch pending state are defined below.

In the dispatch pending state, even in the situation where the task in the RUNNING state should be preempted, the task that should run will not be dispatched. The dispatch for the task that should run will be pending until the system is in a state where dispatching can occur. While dispatching is pending, the task that has been running remains in the RUNNING state, while the task waiting for dispatching remains in the READY state.

Task states during the dispatch pending state can be affected by implementation-specific extensions. More precisely, extensions may allow non-task contexts to invoke service calls that move the task in the RUNNING state to the SUSPENDED state or the DORMANT state. In addition, extensions may allow the service calls to move the invoking task to the SUSPENDED state while in the dispatching disabled state. Task states for these cases are described below.

When the task in the RUNNING state is to be moved to the SUSPENDED state or the DORMANT state, the transition is pending until the system state allows dispatching to occur. While the state transition is pending, the task that has been in the RUNNING state is considered to be in a transitional state. The treatment of a task in this transitional state is implementation-dependent. The task that should be in the RUNNING state remains in the READY state until the dispatch occurs.

[Supplemental Information]

Figure 3-4 explains the task state during the dispatch pending state. Suppose that Task B is activated from the interrupt handler that was invoked by the interrupt that occurred during execution of Task A when the priority of Task B is higher than the priority of Task A. Since the precedence of the interrupt handler is higher than that of the dispatcher, the system is in the dispatch pending state while the interrupt handler is executing. Therefore dispatching does not occur. When the interrupt handler execution terminates, the dispatcher is executed and the task that should run switches from Task A to Task B.

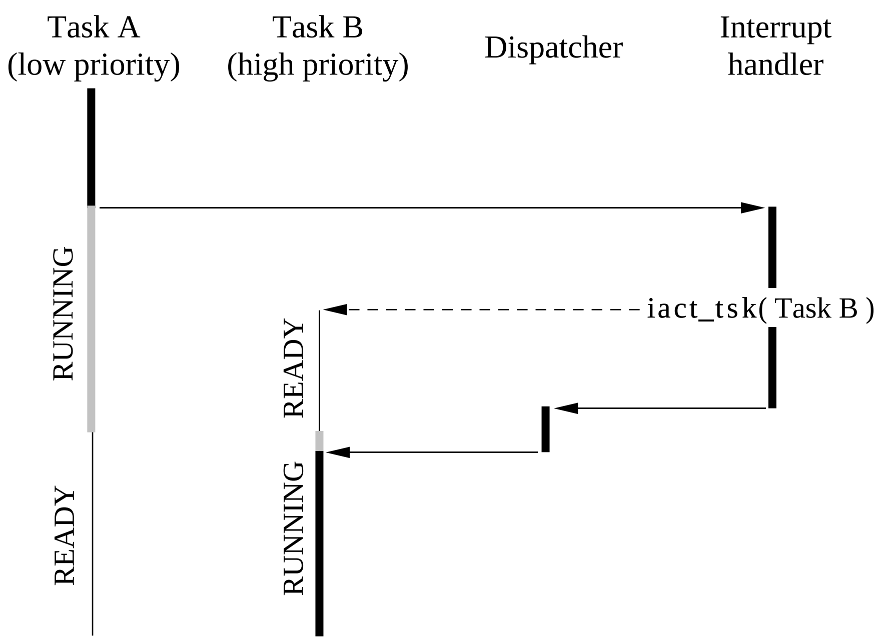


Figure 3-4. Dispatch Pending State and Task States

Even after Task B is activated, Task A is in the RUNNING state and Task B is in the READY state until the dispatcher is started. After the dispatcher executes, Task B is in the RUNNING state, and Task A is in the READY state. Because the dispatcher should be executed atomically, task states during the dispatcher execution are not specified in this specification.

3.6 Service Call Invocation from Non-Task Contexts

3.6.1 Service Calls that can be Invoked from Non-Task Contexts

Service calls that can be invoked from non-task contexts have the letter "i" added to the beginning of their names so they can be distinguished from service calls that can be invoked from task context. Service calls that can be invoked from both non-task contexts and task contexts have a different naming convention as described below. In other words, the service calls are classified into the following three categories:

(a) Service calls for non-task contexts

Service calls whose names begin with "i" are called service calls for non-task contexts. They may be invoked from non-task contexts.

[Supplemental Information]

The following service calls belong to this category:

iact_tsk	activate task
iwup_tsk	wakeup task
irel_wai	release task from waiting
iras_tex	raise task exception handling
isig_sem	release semaphore resource
iset_flg	set eventflag
ipsnd_dtq	send to data queue (polling)
ifsnd_dtq	forced send to data queue
isig_tim	supply time tick
irot_rdq	rotate task precedence
iget_tid	reference task ID in the RUNNING state
iloc_cpu	lock the CPU
iunl_cpu	unlock the CPU

The behavior of the service calls for non-task contexts invoked from task contexts is undefined. When an error should be reported, an E_CTX error is returned.

(b) Service calls that can be invoked from any contexts

Service calls whose names are of the form sns_*yyy* can be invoked from any contexts. They may be invoked from both task contexts and non-task contexts.

[Supplemental Information]

The following service calls belong to this category:

sns_ctx	reference contexts

sns_loc	reference CPU state
sns_dsp	reference dispatching state
sns_dpn	reference dispatch pending state
sns_tex	reference task exception handling state

(c) Service calls for task contexts

The remaining service calls are called service calls for task contexts. They may be invoked from task contexts.

The behavior of the service calls for task contexts invoked from non-task contexts is undefined. When an error should be reported, an E_CTX error is returned.

[Differences from the μITRON3.0 Specification]

Service calls for non-task contexts are specified to have names that begin with "i." Invoking service calls for task contexts from non-task contexts is permitted as an implementation-specific extension.

3.6.2 Delayed Execution of Service Calls

The execution of service calls invoked from non-task contexts may be delayed at most until the processing units that have higher precedence than the dispatcher have terminated. This makes it possible to guarantee the atomicity of service calls without disabling interrupts for too long. This is called delayed execution of service calls.

However, the following service calls are not allowed to have their execution delayed:

iget_tid	reference task ID in the RUNNING state
iloc_cpu	lock the CPU
iunl_cpu	unlock the CPU
sns_ctx	reference contexts
sns_loc	reference CPU state
sns_dsp	reference dispatching state
sns_dpn	reference dispatch pending state
sns_tex	reference task exception handling state

When the service calls have their execution delayed, the processing order of the service calls must correspond to the order in which the service calls were invoked, excluding those service calls that are not allowed to have their execution delayed.

There are situations in which the service calls that are invoked from non-task contexts and that have their execution delayed cannot return some error codes. This is because the detection of some errors depends on the target object's state and the object's state cannot be referenced when the service call's execution is delayed. In these situations, E_OK can be returned instead of the error code that would be returned for non-delayed execution. The error codes that may not be returned when execution is delayed are defined for each service call.

The kernel must store service calls that have their execution delayed. If there is insufficient memory to store a service call for delayed execution, the service call must return an E_NOMEM error.

[Supplemental Information]

The point at which the service call executes after having its execution delayed is up to the implementation as long as the behavior of the delayed execution is the same as described by the specification. A specific case is where service calls invoked during the dispatch pending state may be delayed until the system enters a state where dispatching can occur. Note that there are situations in which iras_tex must be executed even in the dispatching disabled state. See the supplemental information of iras_tex for more details.

When service calls that have their execution delayed return E_OK, it must be guaranteed that those service calls will be executed later.

[Differences from the μITRON3.0 Specification]

The specification regarding delayed execution of service calls has been clarified.

3.6.3 Adding Service Calls that can be Invoked from Non-Task Contexts

When a service call for task contexts with the name *xxx_yyy* (or *zxxx_yyy*) is defined in the μITRON4.0 Specification, an implementation may add a service call for non-task contexts which has the same functionality. In this case the name of the new service call should be i*xxx_yyy* (or i*zxxx_yyy*) regardless of the rule that the names of implementation-specific service calls should begin with the letter “v.” The new service call is still considered to have the same functionality even when some error codes are not returned due to delayed execution of the service call invoked from non-task contexts.

When a service call for task contexts is made invokable from non-task contexts using its original name as an implementation-specific extension, the implementation must also provide a service call where the letter “i” is added at the beginning of its name that is invokable from non-task contexts. On the other hand, when a service call for non-task contexts is made invokable from task contexts using its original name as an implementation-specific extension, the implementation must also provide a service call where the letter “i” is removed from the beginning of its name that is invokable from task contexts.

These rules apply to implementation-specific service calls as well. When there is an implementation-specific service call with the name v*xxx_yyy* and a service call with the same functionality can be invoked from non-task contexts, it must be invokable with the name iv*xxx_yyy*.

3.7 System Initialization Procedure

System initialize procedure is modeled as follows (Figure 3-5):

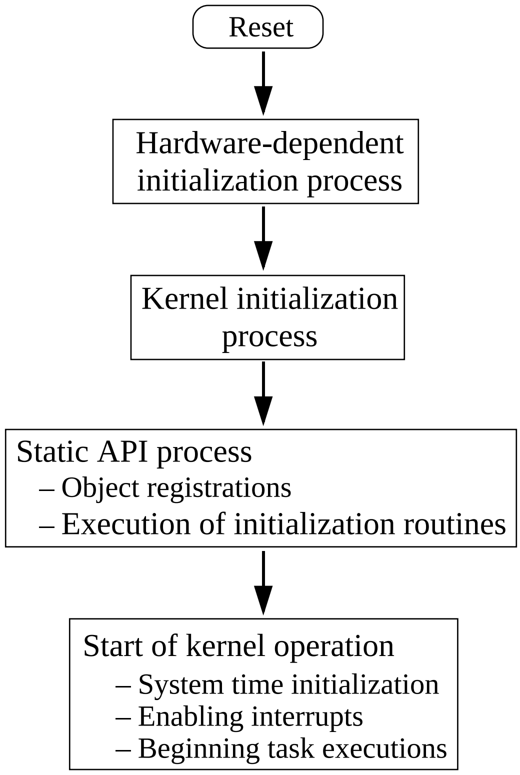


Figure 3-5. System Initialization Procedure

The hardware-dependent initialization process is executed after the system is reset. The application prepares the hardware-dependent initialization process, which is outside of the kernel's control. The kernel initialization process is called at the end of the hardware-dependent initialization process. The method used to call the kernel initialization process is implementation-defined.

Once the kernel initialization process is called, the kernel itself, such as the kernel's internal data structures, is initialized. Then, the static APIs, such as object registrations, are processed. The static APIs, except for ATT_INI, are processed in the order described in the system configuration file. The method used to handle errors detected during the static API processes is implementation-defined.

The processing of the static APIs includes the execution of initialization routines. The initialization routines are prepared by the application and registered with the kernel by using ATT_INI. The initialization routines are executed with all interrupts disabled except for non-kernel interrupts. Disabling non-kernel interrupts is implementation-defined. Allowing initialization routines to invoke service calls and which service calls are invokable are implementation-defined. The initialization routines are executed in the order described with ATT_INI in the system configuration file. The relative order between the execution of initialization routines and the processing of other static APIs is implementation-defined.

After the processing of the static APIs, the kernel operation is started. Specifically, tasks begin execution. At this point interrupts are enabled for the first time and the system time is initialized to 0.

The above description provides only a conceptual model. The real system initialization procedure may be optimized in an implementation-dependent manner as long as the behavior conforms to this conceptual model.

3.8 Object Registration and Release

An object identified by an ID number is registered to the kernel by a static API (CRE_*YYY*) or by a service call (cre_*yyy*) that creates the object. An object is released from the kernel by a service call (del_*yyy*) that deletes the object. After an object is deleted, a new object can be created with the same ID number. When an object is created, the ID number and the necessary information for creating the object are specified. When an object is deleted, the ID number for the object is specified.

The maximum number of objects and the range of the ID numbers that can be registered are implementation-defined. The maximum number of objects that can be created by using service calls and the procedure to designate the range of ID numbers are also implementation-defined.

When a static API (ATT_*YYY*) attaches an object to the kernel, it creates and registers the object without specifying an ID number. Objects registered in this way cannot be referred by ID numbers because the created objects do not have ID numbers, which means that objects created in this way cannot be deleted.

The service call that creates an object and assigns an ID number automatically (acre_*yyy*) assigns the object ID number by selecting an ID number that is not already associated with an object. The ID number assigned to the created object is returned to the application as the return value. The ID number assigned in this way is limited to a positive number because a negative return value from a service call indicates an error occurred. If there is no ID number that can be assigned, the service call returns an E_NOID error.

The method an implementation employs to designate the range of ID numbers available for automatic assignment is implementation-defined. The method used to automatically assign available ID numbers to objects is implementation-dependent.

A synchronization and communication object can be deleted even if there is a task waiting for a condition to be met associated with the object. In this case, the task that is waiting for the condition associated with the deleted object is released from waiting. The service call that placed the task in the WAITING state returns an E_DLT error to the released task. If more than one task is waiting, the tasks are released from waiting in the order in which they reside in the wait queue for the synchronization and communication object. Therefore, among tasks with the same priority that are moved into the READY state, tasks closer to the head of the wait queue have higher precedence. In case the synchronization and communication object has multiple wait queues, the order that tasks are released from different wait queues is implementation-dependent.

[Standard Profile]

The Standard Profile requires an implementation to support at least ID numbers from 1 to 255. Also, the Standard Profile requires that at least 255 objects can be registered for objects that are referenced by ID numbers and are part of the Standard Profile.

3.9 Description Format for Processing Unit

The µITRON4.0 Specification specifies the format for writing each of the following processing units in the C language: interrupt service routines, time event handlers (cyclic handlers, alarm handlers, overrun handlers), extended service call routines, tasks, and task exception handling routines. If TA_HLNG (processing unit started through a high-level language interface) is specified as the object attribute when the processing unit is registered with the kernel, the processing unit is started assuming it is written in the specified format.

On the other hand, the µITRON4.0 Specification does not specify the format for writing processing units in assembly language. If TA_ASM (processing unit started through an assembly language interface) is specified as the object attribute when the processing unit is registered with the kernel, the processing unit is started assuming it is written in the format specified by the implementation.

The format for writing interrupt handlers and CPU exception handlers and the object attributes used to register them with the kernel are implementation-defined and are not specified in the µITRON4.0 Specification.

[Supplemental Information]

The µITRON4.0 Specification does not specify the service call that returns from interrupt handlers (ret_int in the previous specifications). This is not because the process executed by ret_int in the previous specifications is no longer needed, but rather because how to write interrupt handlers is now implementation-defined. There may be a case where a service call corresponding to ret_int may be provided by an implementation. This also applies to returning from CPU exception handlers.

The µITRON4.0 Specification does not specify the service call that returns from time event handlers (ret_tmr in the previous specifications). This is not because the process executed by ret_tmr in the previous specifications is no longer needed, but rather because it is now possible to return simply from time event handlers written in the C language. There may be a case where a service call corresponding to ret_tmr is provided by an implementation in order to return from time event handlers written in assembly language. This also applies to returning from interrupt service routines, extended service call routines, and task exception handling routines.

[Differences from the μITRON3.0 Specification]

The μITRON4.0 Specification specifies the format for writing each processing unit in the C language, but does not specify service calls (ret_*yyy*) for returning from processing units, because they are only needed when the processing units are written in assembly language.

3.10 Kernel Configuration Constants and Macros

Applications use kernel configuration constants and macros to reference the kernel configuration in order to improve application program portability. The method used to define kernel configuration constants and macros is implementation-dependent as long as they can be referenced from an application program

Kernel configuration constants and macros are not defined when functions related to them are not supported.

[Supplemental Information]

Kernel configuration constants and macros may be defined as fixed values in kernel header files or may be generated by a configurator. Alternatively they may be defined in header files prepared by the application and then used to configure the kernel.

[Differences from the μITRON3.0 Specification]

The μITRON4.0 Specification newly introduces kernel configuration constants and macros.

3.11 Kernel Common Definitions

3.11.1 Kernel Common Constants

(1) Object Attributes

TA_HLNG	0x00	Start a processing unit through a high-level language interface
TA_ASM	0x01	Start a processing unit through an assembly language interface
TA_TFIFO	0x00	Task wait queue is in FIFO order
TA_TPRI	0x01	Task wait queue is in task priority order

TA_MFIFO	0x00	Message queue is in FIFO order
TA_MPRI	0x02	Message queue is in message priority order

[Differences from the μITRON3.0 Specification]

The values of TA_HLNG and TA_ASM have been exchanged.

(2) Main Error Codes Used in Kernel

The kernel uses the main error codes specified in Section 2.3.2, except for the three codes, E_CLS, E_WBLK, and E_BOVR.

[Standard Profile]

In the Standard Profile the following main error codes are generated and must be detected:

E_OBJ	Object state error
E_QOVR	Queue overflow
E_RLWAI	Forced release from waiting
E_TMOUT	Polling failure or timeout

Applications that need to be portable across kernels adhering to the Standard Profile must not depend on detecting errors beyond those listed above.

[Supplemental Information]

In the Standard Profile the following main error codes are not generated or need not be detected:

(a) Error codes not used by the kernel

E_CLS, E_WBLK, E_BOVR

(b) Error codes not generated by Standard Profile functions

E_OACV, E_NOID, E_NOEXS, E_DLT

(c) Error codes that are implementation-dependent

E_SYS, E_RSFN, E_NOMEM

(d) Error codes whose detection can be omitted

E_NOSPT, E_RSATR, E_PAR, E_ID, E_CTX, E_MACV, E_ILUSE

(3) Service Call Function Codes

Function codes ranging from (–0xe0) to (–0x05) are assigned to kernel service calls. However, a function code is not assigned to `cal_svc`. The assignment of function codes is specified in each function in Chapter 4.

Function codes within the range from (–0xe0) to (–0x05) that are not assigned in the specification are reserved for the kernel function extensions in the future. Function codes ranging from (–0x100) to (–0xe1) can be used for implementation-specific service calls. Function codes ranging from (–0x200) to (–0x101) are reserved for kernel function extensions in the future. However, they can be used for implementation-specific service calls if needed.

[Differences from the µITRON3.0 Specification]

The values of function codes have been reassigned.

[Rationale]

Function codes of service calls included in the Standard Profile range from (–0x80) to (–0x05) in order to fit within 8 bits.

(4) Other Kernel Common Constants

TSK_SELF	0	Specifying invoking task
TSK_NONE	0	No applicable task

[Differences from the µITRON3.0 Specification]

TSK_NONE has been added. In the µITRON3.0 Specification, FALSE (= 0) was used when there was no applicable task available.

3.11.2 Kernel Common Configuration Constants

(1) Priority Range

TMIN_TPRI	Minimum task priority (= 1)
TMAX_TPRI	Maximum task priority
TMIN_MPRI	Minimum message priority (= 1)
TMAX_MPRI	Maximum message priority

[Standard Profile]

These kernel configuration constants must be defined in the Standard Profile. TMAX_TPRI must not be less than 16 and TMAX_MPRI must not be less than TMAX_TPRI.

[Supplemental Information]

Although TMIN_TPRI and TMIN_MRI are fixed as 1 in this specification, implementation-specific extensions may configure the kernel to use a value other than 1.

(2) Version Information

TKERNEL_MAKER	Kernel maker code
TKERNEL_PRID	Identification number of the kernel
TKERNEL_SPVER	Version number of the ITRON Specification
TKERNEL_PRVER	Version number of the kernel

[Standard Profile]

These kernel configuration constants must be defined in the Standard Profile.

[Supplemental Information]

See the functional description of ref_ver for the constant values that represent version information.

Chapter 4 µITRON4.0 Functions

4.1 Task Management Functions

Task management functions provide direct control of task states and reference to the task states. Task management functions include the ability to create and delete a task, to activate and terminate a task, to cancel activation requests, and to reference the state of a task. A task is an object identified by an ID number. The ID number of a task is called the task ID. See Section 3.2 for rules governing task scheduling and state transitions.

A task has a base priority and a current priority for controlling the order of task execution. In this specification, the words “task priority” refer to the task’s current priority. When the task is activated, the base priority is set to the task’s initial priority as defined when the task is created. If mutexes are not used, the current priority and the base priority are always equal. Therefore, the current priority of a task is set to its initial priority when the task is activated. For more information about how mutexes change the current priority, see Section 4.5.1.

Activation requests for a task are queued. In other words, if a task has already been activated and an activation request is made for the task, the new request is recorded. When the task terminates under this situation, the task will be automatically activated again. However, activation requests will not be queued when the service call that activates a task with the specified start code (sta_tsk) is used. A task includes an activation request count to realize the activation request queuing. This count is cleared to 0 when the task is created.

When a task is activated, its extended information (exinf) is passed as a parameter. However, when a task is activated by the service call with a start code (sta_tsk), the specified start code is passed through the parameter instead of the extended information.

When a task terminates, the kernel does not release resources that the task acquired such as semaphore resources and memory blocks. However, the kernel unlocks mutexes acquired by the task. The application program is responsible for releasing resources acquired by the task when the task terminates.

The following actions must be taken when creating, activating, terminating, and deleting a task. When a task is created, the activation request count is cleared, the task’s exception handling routine is set to undefined (see Section 4.3), and the task’s processor time limit is set to undefined (see Section 4.7.4). When a task is activated, the task’s base priority and current priority are initialized, the task’s wakeup request count is cleared (see Section 4.2), the task’s suspension count is cleared (see Section 4.2), the task’s pending exception code is cleared (see Section 4.3), and the task’s exception handling is disabled (see Section 4.3). When a task is terminated, all mutexes locked by the task are unlocked (see Section 4.5.1) and the processor time limit is set to undefined (see Section 4.7.4). When a task is deleted, the task's stack space is released if the stack space was allocated by the kernel when the task was created.

The format to write a task in the C language is shown below:

```
void task ( VP_INT exinf )
{
        /* Body of the task */
        ext_tsk ( ) ;
}
```

The behavior of a task returning from its main routine is identical to invoking `ext_tsk`, i.e. the task terminates. `exd_tsk` deletes the invoking task in addition to terminating the task.

The following kernel configuration constant is defined for use with task management functions:

TMAX_ACTCNT Maximum activation request count

The following data type packets are defined for creating and referencing tasks:

```
typedef struct t_ctsk {
        ATR         tskatr ;     /* Task attribute */
        VP_INT      exinf ;      /* Task extended information */
        FP          task ;       /* Task start address */
        PRI         itskpri ;    /* Task initial priority */
        SIZE        stksz ;      /* Task stack size (in bytes) */
        VP          stk ;        /* Base address of task stack space */
        /* Other implementation specific fields may be added. */
} T_CTSK ;

typedef struct t_rtsk {
        STAT        tskstat ;    /* Task state */
        PRI         tskpri ;     /* Task current priority */
        PRI         tskbpri ;    /* Task base priority */
        STAT        tskwait ;    /* Reason for waiting */
        ID          wobjid ;     /* Object ID number for which the task is
                                    waiting */
        TMO         lefttmo ;    /* Remaining time until timeout */
        UINT        actcnt ;     /* Activation request count */
        UINT        wupcnt ;     /* Wakeup request count */
        UINT        suscnt ;     /* Suspension count */
        /* Other implementation specific fields may be added. */
} T_RTSK ;

typedef struct t_rtst {
        STAT        tskstat ;    /* Task state */
        STAT        tskwait ;    /* Reason for waiting */
        /* Other implementation specific fields may be added. */
} T_RTST ;
```

The following represents the function codes for the task management service calls:

TFN_CRE_TSK	–0x05	Function code of cre_tsk
TFN_ACRE_TSK	–0xc1	Function code of acre_tsk
TFN_DEL_TSK	–0x06	Function code of del_tsk
TFN_ACT_TSK	–0x07	Function code of act_tsk
TFN_IACT_TSK	–0x71	Function code of iact_tsk
TFN_CAN_ACT	–0x08	Function code of can_act
TFN_STA_TSK	–0x09	Function code of sta_tsk
TFN_EXT_TSK	–0x0a	Function code of ext_tsk
TFN_EXD_TSK	–0x0b	Function code of exd_tsk
TFN_TER_TSK	–0x0c	Function code of ter_tsk
TFN_CHG_PRI	–0x0d	Function code of chg_pri
TFN_GET_PRI	–0x0e	Function code of get_pri
TFN_REF_TSK	–0x0f	Function code of ref_tsk
TFN_REF_TST	–0x10	Function code of ref_tst

[Standard Profile]

The Standard Profile requires support for task management functions except for dynamic creation and deletion of a task (cre_tsk, acre_tsk, del_tsk), activation of a task with the specified start code (sta_tsk), termination and deletion of a task (exd_tsk), and reference of a task state (ref_tsk, ref_tst).

The Standard Profile requires support for an activation request count of one or more. Therefore, TMAX_ACTCNT must be at least 1.

[Supplemental Information]

The contexts and states under which tasks execute are summarized as follows:

TMAX_ACTCNT must be 0 if activation request queuing of a task is not supported.

- Tasks execute in their own independent contexts (see Section 3.5.1). The contexts in which tasks execute are classified as task contexts (see Section 3.5.2).
- Tasks execute at lower precedence than the dispatcher (see Section 3.5.3).
- After tasks start, the system is both in the CPU unlocked state and in the dispatching enabled state. When tasks exit, the system must be both in the CPU unlocked state and in the dispatching enabled state (see Sections 3.5.4 and 3.5.5).

TMAX_ACTCNT must be 0 if activation request queuing of a task is not supported.

[Differences from the μITRON3.0 Specification]

Functions that directly operate on tasks and that have no relation with waiting states are classified as task management functions. Functions that change task precedence (rot_rdq), reference the ID of the task in the RUNNING state (get_tid), and enable or disable task dispatching (ena_dsp, dis_dsp) are now classified as system state management functions. The function releasing a task from a waiting state (rel_wai) is now classified as a task dependent synchronization function.

Service calls for requesting task activation and canceling the activation requests have been added (act_tsk, can_tsk). The service call for starting a task with the specified start code (sta_tsk) has not been removed to maintain backward compatibility with μITRON3.0; however, this service call is not required in the Standard Profile.

The concept of task base priorities is introduced due to the addition of mutexes. If mutexes are not used, the behavior is the same as in μITRON3.0 because the base priority is always equal to the current priority.

Returning from a task's main routine now terminates the task.

CRE_TSK	Create Task (Static API)	[S]
cre_tsk	Create Task	
acre_tsk	Create Task (ID Number Automatic Assignment)	

[Static API]

CRE_TSK (ID tskid, { ATR tskatr, VP_INT exinf, FP task,
PRI itskpri, SIZE stksz, VP stk }) ;

[C Language API]

ER ercd = cre_tsk (ID tskid, T_CTSK *pk_ctsk) ;
ER_ID tskid = acre_tsk (T_CTSK *pk_ctsk) ;

[Parameter]

ID	tskid	ID number of the task to be created (except acre_tsk)
T_CTSK *	pk_ctsk	Pointer to the packet containing the task creation information (In CRE_TSK, the contents must be directly specified.)

pk_ctsk includes (T_CTSK type)

ATR	tskatr	Task attribute
VP_INT	exinf	Task extend information
FP	task	Task start address
PRI	itskpri	Task initial priority
SIZE	stksz	Task stack size (in bytes)
VP	stk	Base address of task stack space

(Other implementation specific information may be added.)

[Return Parameter]

cre_tsk:

ER	ercd	E_OK for normal completion or error code

acre_tsk:

ER_ID	tskid	ID number (positive value) of the created task or error code

[Error Code]

E_ID	Invalid ID number (tskid is invalid or unusable; only cre_tsk)
E_NOID	No ID number available (there is no task ID assignable; only acre_tsk)
E_NOMEM	Insufficient memory (stack space or other memory cannot be allocated)
E_RSATR	Reserved attribute (tskatr is invalid or unusable)

E_PAR	Parameter error (pk_ctsk, task, itskpri, stksz, or stk is invalid)
E_OBJ	Object state error (task is already registered; only cre_tsk)

[Functional Description]

These service calls create a task with an ID number specified by tskid based on the information contained in the packet pointed to by pk_ctsk. The task is moved from the NON-EXISTENT state to either the DORMANT state or the READY state. In addition, the actions that must be taken at task creation time are performed. tskatr is the attribute of the task. exinf is the extended information passed as a parameter to the task when the task is started. task is the start address of the task. itskpri is the initial value of the task's base priority when the task is activated. stksz is the stack size in bytes of the task. stk is the base address of the task's stack space.

In CRE_TSK, tskid is an integer parameter with automatic assignment. tskatr is a preprocessor constant expression.

acre_tsk assigns a task ID from the pool of unassigned task IDs and returns the assigned task ID.

tskatr can be specified as ((TA_HLNG || TA_ASM) | [TA_ACT]). If TA_HLNG (= 0x00) is specified, the task is started through the C language interface. If TA_ASM (= 0x01) is specified, the task is started through the assembler language interface. After the creation, the task is moved to the READY state if TA_ACT (= 0x02) is specified, and is moved to the DORMANT state otherwise.

The memory area defined by the base address stk and the size stksz is used by the task for its stack space during execution. If stk is NULL (= 0), the kernel allocates a memory area with size stksz for use as the task's stack space.

[Standard Profile]

The Standard Profile does not require support for when TA_ASM is specified in tskatr. It also does not require support for when other values than NULL are specified in stk.

[Supplemental Information]

Several processing units besides the task, such as service calls invoked by the task and interrupt handlers started during the task's execution, may use the task's stack space depending on the implementation. The implementation's documentation, such as the product manuals, should describe how to calculate the necessary stack size.

The base address of the task stack's space indicates the lowest address of the memory area used as the task stack space. Therefore, in general, the initial value of the task's stack pointer does not correspond to the base address of the stack.

A task cannot specify its own task ID in tskid. If a task does specify its own task ID, cre_tsk returns an E_OBJ error because the task is already registered.

[Differences from the μITRON3.0 Specification]

The base address of the task's stack space, stk, has been added. stk should be set to NULL if compatibility is required with μITRON3.0.

The order of tskatr and exinf in the task's creation information packet has been exchanged. The data type of exinf has been changed from VP to VP_INT. The data type of stksz has been changed from INT to SIZE.

The ability to move a task directly to the READY state has been added through the use of the added task attribute TA_ACT. This is useful for the case when a task is statically created. The task attributes indicating the task uses co-processors have been removed. When necessary, such attributes can be added as implementation-specific.

acre_tsk has been newly added.

del_tsk Delete Task

[C Language API]

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

[Parameter]

ID	tskid	ID number of the task to be deleted

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (tskid is invalid or unusable)
E_NOEXS	Non-existent object (specified task is not registered)
E_OBJ	Object state error (specified task is not in the DORMANT state)

[Functional Description]

This service call deletes the task specified by tskid. The deleted task is moved from the DORMANT state to the NON-EXISTENT state and the actions that must be taken at task deletion time are performed.

If the task is not in the DORMANT state, an E_OBJ error is returned. However, if the task is not registered, an E_NOEXS error is returned.

[Supplemental Information]

A task cannot specify its own task ID in tskid. If a task does specify its own task ID, del_tsk returns an E_OBJ error because the task is not in the DORMANT state. exd_tsk can be used by a task to terminate and delete itself.

act_tsk Activate Task [S]
iact_tsk [S]

[C Language API]

```
ER ercd = act_tsk ( ID tskid ) ;
ER ercd = iact_tsk ( ID tskid ) ;
```

[Parameter]

ID	tskid	ID number of the task to be activated

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (tskid is invalid or unusable)
E_NOEXS	Non-existent object (specified task is not registered)
E_QOVR	Queue overflow (overflow of activate request queuing count)

[Functional Description]

These service calls activate the task specified by tskid. The task is moved from the DORMANT state to the READY state and the actions that must be taken at task activation time are performed. The extended information of the task is passed to the task as a parameter.

If the task is not in the DORMANT state, the activation request for the task is queued. (However, if the task in the NON-EXISTENT state, an E_NOEXS error is returned.) Specifically, the activation request count is incremented by 1. If the count then exceeds the maximum possible count, an E_QOVR error is returned.

If the service call is invoked from non-task contexts and has its execution delayed, an E_QOVR error may not be returned.

If tskid is TSK_SELF (= 0), the invoking task is specified. If TSK_SELF is specified when this service call is invoked from non-task contexts, an E_ID error is returned.

[Supplemental Information]

The Standard Profile requires the maximum activation request count to be at least 1. This implies that a kernel that is compatible with the Standard Profile may not always return an E_QOVR error even if these service calls are invoked on a task with queued activation requests.

[Differences from the μITRON3.0 Specification]

These service calls have been newly added.

can_act Cancel Task Activation Requests [S]

[C Language API]

```
ER_UINT actcnt = can_act ( ID tskid ) ;
```

[Parameter]

ID	tskid	ID number of the task for cancelling activation requests

[Return Parameter]

ER_UINT	actcnt	Activation request count (positive value or 0) or error code

[Error Code]

E_ID	Invalid ID number (tskid is invalid or unusable)
E_NOEXS	Non-existent object (specified task is not registered)

[Functional Description]

This service call cancels all queued activation requests for the task specified by tskid and returns the cancelled request count for the task. Specifically, the activation request count for the task is cleared to 0. The value returned is the count before it was cleared.

If tskid is TSK_SELF (= 0), the invoking task is specified.

[Supplemental Information]

This service call may specify a task in the DORMANT state. In this case, the service call returns a count of 0 because activation requests are not queued for the task.

This service call can be used to check if a task completes a process within a cycle correctly when the task is activated cyclically. Specifically, can_act should be invoked when the task completes the process. A return value of 1 or more from can_act indicates that the next activation is requested before the task completes the process in the previous cycle. The task can take measures for this case.

[Differences from the μITRON3.0 Specification]

This service call has been newly added.

sta_tsk Activate Task (with a Start Code)

[C Language API]

```
ER ercd = sta_tsk ( ID tskid, VP_INT stacd ) ;
```

[Parameter]

ID	tskid	ID number of the task to be activated
VP_INT	stacd	Start code of the task

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (tskid is invalid or unusable)
E_NOEXS	Non-existent object (specified task is not registered)
E_OBJ	Object state error (specified task is not in the DORMANT state)

[Functional Description]

This service call activates the task specified by tskid. The task is moved from the DORMANT state to the READY state and the actions that must be taken at task activation time are performed. The start code, stacd, is passed to the task as a parameter.

If the task is not in the DORMANT state, the service call does not queue a request for activation and returns an E_OBJ error. If the task is in the NON-EXISTENT state, it returns an E_NOEXS error.

[Supplemental Information]

A task cannot specify its own task ID in tskid. If a task does specify its own task ID, sta_tsk returns an E_OBJ error because the task is not in the DORMANT state.

[Differences from the µITRON3.0 Specification]

The data type for stacd has been changed from INT to VP_INT.

ext_tsk Terminate Invoking Task [S]

[C Language API]

```
void ext_tsk ( ) ;
```

[Parameter]

None

[Return Parameter]

This service call does not return.

[Functional Description]

This service call terminates the invoking task. The invoking task is moved from the RUNNING state to the DORMANT state and the actions that must be taken at task termination time are performed.

If activation requests are queued, that is, if the activation request count for the invoking task is 1 or more, the count is decremented by 1 and the task is moved to the READY state. In this case, the actions that must be taken at task activation time are performed. The extended information of the task is passed to the task as a parameter.

This service call never returns; therefore, no error code is returned even if an error is encountered in the service call. The behavior of the service call when an error is detected is implementation-defined.

[Supplemental Information]

When activation requests are queued for the invoking task, this service call will reactivate the task after it has been terminated. This implies that all mutexes locked by the task are unlocked and the processing time limit is set to undefined. In addition the base priority, the current priority, the wakeup request count, the suspension count, the pending exception code, and the task exception handling state are all reset to their initial values. The task has the lowest precedence among all tasks with the same priority in the READY state.

When an error is detected in the service call, the information regarding the error can be logged.

The behavior of a task returning from its main routine is identical to invoking ext_tsk.

[Differences from the μITRON3.0 Specification]

Tasks that invoke ext_tsk may be reactivated due to the addition of the activation request count.

exd_tsk Terminate and Delete Invoking Task

[C Language API]

```
void exd_tsk ( ) ;
```

[Parameter]

None

[Return Parameter]

This service call does not return.

[Functional Description]

This service call terminates and deletes the invoking task. The task is moved from the RUNNING state to the NON-EXISTENT state and the actions that must be taken at task termination and deletion time are performed.

This service call never returns; therefore, no error code is returned even if an error is encountered in the service call. The behavior of the service call when an error is detected is implementation-defined.

[Supplemental Information]

This service call terminates and deletes the invoking task even if activation requests are queued for the invoking task. The activation request count has no meaning when the task is in the NON-EXISTENT state.

When an error is detected in the service call, the information regarding the error can be logged.

ter_tsk Terminate Task [S]

[C Language API]

ER ercd = ter_tsk (ID tskid) ;

[Parameter]

ID	tskid	ID number of the task to be terminated

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (tskid is invalid or unusable)
E_NOEXS	Non-existent object (specified task is not registered)
E_ILUSE	Illegal service call use (specified task is an invoking task)
E_OBJ	Object state error (specified task is in the DORMANT state)

[Functional Description]

This service call terminates the task specified by tskid. The task is moved to the DORMANT state and the actions that must be taken at task termination time are performed.

If activate requests are queued, that is, if the activation request count for the specified task is 1 or more, the count is decremented by 1 and the task is moved to the READY state. In this case, the actions that must be taken at task activation time are performed. The extended information of the task is passed to the task as a parameter.

If the task is in the DORMANT state, an E_OBJ error is returned. A task cannot terminate itself with this service call. If a task specifies its own task ID in tskid, an E_ILUSE error is returned.

[Supplemental Information]

This service call forces the specified task to terminate even if the task is in the blocked state. When the task is waiting in a wait queue, the task is removed from the queue. In this case, some other tasks that are in the wait queue may need to be released from waiting. See the functional descriptions of snd_mbf and get_mpl.

When activation requests are queued for the specified task, this service call will reactivate the task after it has been terminated. This implies that all mutexes locked by the task are unlocked and the processing time limit is set to undefined. In addition the base priority, the current priority, the wakeup request count, the suspension count, the pending exception code, and the task exception handling state are all reset to their initial values. The task has the lowest precedence among all tasks with the same priority in the READY state.

[Differences from the µITRON3.0 Specification]

The main error code when the invoking task is specified has been changed from E_OBJ to E_ILUSE.

This service call may reactivate the specified task due to the addition of the activation request count.

chg_pri Change Task Priority [S]

[C Language API]

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

[Parameter]

ID	tskid	ID number of the task whose priority is to be changed
PRI	tskpri	New base priority of the task

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (tskid is invalid or unusable)
E_NOEXS	Non-existent object (specified task is not registered)
E_PAR	Parameter error (tskpri is invalid)
E_ILUSE	Illegal service call use (priority ceiling violation)
E_OBJ	Object state error (specified task is in the DORMANT state)

[Functional Description]

This service call changes the base priority of the task specified by tskid to the priority value specified by tskpri. The current priority can also be changed.

If tskid is TSK_SELF (= 0), the priority of the invoking task is changed. If tskpri is TPRI_INI (= 0), the base priority is changed to the task's initial priority.

If the invocation of this service call results in equal base and current priorities, which is always the case if mutexes are not used, the following actions are performed. If the task is runnable, the task's precedence is changed to reflect the new priority. The task will have the lowest precedence among tasks with the same priority. If the task is waiting in a wait queue, the task's position in the queue is changed to reflect the new priority. The task will be placed last among tasks with the same priority.

If the task locked mutexes with the TA_CEILING attribute and the new base priority, tskpri, is higher than one of the ceilings of the mutexes, an E_ILUSE error is returned.

[Supplemental Information]

When the task is waiting in a wait queue, this service call may change the task's order in the wait queue. In this case, some other tasks that are in the wait queue may need to be released from waiting. See the functional descriptions of snd_mbf and get_mpl.

If the specified task is waiting for a mutex with the TA_INHERIT attribute, transitive priority inheritance needs to be applied as the result of changing the base priority of the task.

When mutexes are not used and when this service call is invoked with the invoking task in tskid and its base priority in tskpri, the task will have the lowest precedence among all tasks with the same priority. Therefore, this service call can be used to yield the execution privilege to another task.

[Differences from the μITRON3.0 Specification]

chg_pri now changes the base priority of a task due to the addition of mutexes. Allowing tskpri to be set to TPRI_INI is now standard.

get_pri Reference Task Priority [S]

[C Language API]

```
ER ercd = get_pri ( ID tskid, PRI *p_tskpri ) ;
```

[Parameter]

ID	tskid	ID number of the task to reference

[Return Parameter]

ER	ercd	E_OK for normal completion or error code
PRI	tskpri	Current priority of specified task

[Error Code]

E_ID	Invalid ID number (tskid is invalid or unusable)
E_NOEXS	Non-existent object (specified task is not registered)
E_OBJ	Object state error (specified task is in the DORMANT state)

[Functional Description]

This service call returns the current priority of the task specified by tskid through tskpri.

If tskid is TSK_SELF (= 0), the current priority of the invoking task is returned.

[Supplemental Information]

get_pri refers to the task's current priority while chg_pri changes the task's base priority.

[Differences from the µITRON3.0 Specification]

This service call has been newly added, because a method is required to obtain an invoking task's priority with minimal overhead when the priority of a message to be sent should be set to the task's priority.

[Rationale]

The priority is returned through tskpri as opposed to a return value in order for the service call to be consistent with other similar service calls (get_*yyy*) and in order to allow an implementation-specific extension to use negative values for priorities.

ref_tsk Reference Task State

[C Language API]

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

[Parameter]

ID	tskid	ID number of the task to be referenced
T_RTSK *	pk_rtsk	Pointer to the packet returning the task state

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

pk_rtsk includes (T_RTSK type)

STAT	tskstat	Task state
PRI	tskpri	Task current priority
PRI	tskbpri	Task base priority
STAT	tskwait	Reason for waiting
ID	wobjid	Object ID number for which the task is waiting
TMO	lefttmo	Remaining time until timeout
UINT	actcnt	Activation request count
UINT	wupcnt	Wakeup request count
UINT	suscnt	Suspension count

(Other implementation specific information may be added.)

[Error Code]

E_ID	Invalid ID number (tskid is invalid or unusable)
E_NOEXS	Non-existent object (specified task is not registered)
E_PAR	Parameter error (pk_rtsk is invalid)

[Functional Description]

This service call references the state of the task specified by tskid. The state of the task is returned through the packet pointed to by pk_rtsk. If the specified task is in the NON-EXISTENT state, an E_NOEXS error is returned.

One of the following codes is returned through tskstat to indicate the state of the task:

TTS_RUN	0x01	RUNNING state
TTS_RDY	0x02	READY state
TTS_WAI	0x04	WAITING state
TTS_SUS	0x08	SUSPENDED state
TTS_WAS	0x0c	WAITING-SUSPENDED state
TTS_DMT	0x10	DORMANT state

If the task is not in the DORMANT state, the current priority is returned through tskpri and the base priority is returned through tskbpri. If the task is in the DORMANT state, the values returned through tskpri and tskbpri are implementation-dependent.

If the task is in the WAITING state, including the WAITING-SUSPENDED state, one of the following codes is returned through tskwait to indicate the reason of the task's waiting. If the task is not in the WAITING state, the value returned through tskwait is implementation-dependent.

TTW_SLP	0x0001	Sleeping state
TTW_DLY	0x0002	Delayed state
TTW_SEM	0x0004	Waiting state for a semaphore resource
TTW_FLG	0x0008	Waiting state for an eventflag
TTW_SDTQ	0x0010	Sending waiting state for a data queue
TTW_RDTQ	0x0020	Receiving waiting state for a data queue
TTW_MBX	0x0040	Receiving waiting state for a mailbox
TTW_MTX	0x0080	Waiting state for a mutex
TTW_SMBF	0x0100	Sending waiting state for a message buffer
TTW_RMBF	0x0200	Receiving waiting state for a message buffer
TTW_CAL	0x0400	Calling waiting state for a rendezvous
TTW_ACP	0x0800	Accepting waiting state for a rendezvous
TTW_RDV	0x1000	Terminating waiting state for a rendezvous
TTW_MPF	0x2000	Waiting state for a fixed-sized memory block
TTW_MPL	0x4000	Waiting state for a variable-sized memory block

If the task is in the WAITING state, including the WAITING-SUSPENDED state, the ID of the object the task is waiting for is returned through wobjid. This does not apply when the task is in the sleeping state, the delayed state, or the termination waiting state for a rendezvous. In these states, the value returned through wobjid is implementation-dependent. If the task is not in the WAITING state, the value returned through wobjid is also implementation-dependent.

When the task is in the WAITING state, including the WAITING-SUSPENDED state, but not in the delayed state, the amount of time remaining for the task to timeout is returned through the parameter lefttmo. The value of lefttmo is calculated by subtracting the current time from the time at which the task will timeout. The value returned through lefttmo, however, must be less than or equal to the actual amount of time until timeout. This means that if the timeout happens at the next time tick, 0 is returned through lefttmo. If the task is in the WAITING state forever (that is, waiting without a timeout), TMO_FEVR is returned through lefttmo. If the task is not in the WAITING state or is in the delayed state, the value returned through lefttmo is implementation-dependent.

The service call returns the task's activation request count through actcnt.

If the task is not in the DORMANT state, the wakeup request count and suspension count are returned through wupcnt and suscnt respectively. If the task is in the DORMANT state, the values returned through wupcnt and suscnt are implementation-dependent.

If tskid is TSK_SELF (= 0), the state of the invoking task is referenced.

[Differences from the µITRON3.0 Specification]

Referencing many pieces of information in the µITRON3.0 specification was implementation-dependent, but is now considered standard. The return parameter wid has been changed to wobjid. In addition the following items have been added: task base priority (tskbpri), remaining time until timeout (lefttmo), and activation request count (actcnt). The extended information has been removed from the list.

The order of tskstat and tskpri in pk_rtsk has been exchanged. The data type for tskstat has been changed from UINT to STAT. The order of parameters and of return parameters has been changed.

The return values are now implementation-dependent under cases where parameters have no meaning for specific tasks states. For example, if the task is in the DORMANT state, the value returned through tskpri is implementation-dependent.

The values returned through tskwait have been reassigned.

[Rationale]

If the task is in the delayed state, the value returned through lefttmo is implement-dependent because the delayed time data type used by dly_tsk is RELTIM (unsigned integer) and it cannot be always returned through lefttmo which is of TMO (signed integer).

ref_tst Reference Task State (Simplified Version)

[C Language API]

ER ercd = ref_tst (ID tskid, T_RTST *pk_rtst) ;

[Parameter]

ID	tskid	ID number of the task to be referenced
T_RTST *	pk_rtst	Pointer to the packet returning the task state

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

pk_rtst includes (T_RTST type)

STAT	tskstat	Task state
STAT	tskwait	Reason for waiting

(Other implementation specific information may be added.)

[Error Code]

E_ID	Invalid ID number (tskid is invalid or unusable)
E_NOEXS	Non-existent object (specified task is not registered)
E_PAR	Parameter error (pk_rtst is invalid)

[Functional Description]

This service call references the minimum task state information for the task specified by tskid. The state of the task is returned through the packet pointed to by pk_rtst.

This service call is a simplified version of ref_tsk. The same values returned by ref_tsk through tskstat and tskwait apply to ref_tst as well.

If tskid is TSK_SELF (= 0), the state of the invoking task is referenced.

[Rationale]

A task's information can be referenced with ref_tsk. However, if only minimum information is required, an overhead on data space is incurred for the rest of the possible information. A new service call, ref_tst, has been added in order to extract just the minimum task information.

[Differences from the µITRON3.0 Specification]

This service call has been newly added.

4.2 Task Dependent Synchronization Functions

Task dependent synchronization functions provide direct control of task states to synchronize tasks. Task dependent synchronization functions include the ability to put a task to the sleeping state, to wakeup a task from the sleeping state, to cancel wakeup requests, to forcibly release a task from waiting, to suspend a task, to resume a task from the SUSPENDED state, and to delay the execution of the invoking task.

Wakeup requests for a task are queued. In other words, if a task is not in the sleeping state and a wakeup request is made for the task, the new request is recorded. When the task enters the sleeping state under this situation, the task will not be put in the sleeping state. A task includes a wakeup request count to realize the wakeup request queuing. This count is cleared to 0 when the task is activated.

Suspension requests for a task are nested. In other words, if a task has already been in the SUSPENDED state, including WAITING-SUSPENDED state, and an attempt is made to put the task in the SUSPENDED state again, the new request is recorded. When an attempt is made to resume the task from SUSPENDED state under this situation, the task will not be resumed. A task includes a suspension count to realize the suspension request nesting. This count is cleared to 0 when the task is activated.

The following kernel configuration constants are defined for use with task dependent synchronization functions:

TMAX_WUPCNT	Maximum wakeup request count
TMAX_SUSCNT	Maximum suspension count

The following represents the function codes for the task dependent synchronization service calls:

TFN_SLP_TSK	–0x11	Function code of slp_tsk
TFN_TSLP_TSK	–0x12	Function code of tslp_tsk
TFN_WUP_TSK	–0x13	Function code of wup_tsk
TFN_IWUP_TSK	–0x72	Function code of iwup_tsk
TFN_CAN_WUP	–0x14	Function code of can_wup
TFN_REL_WAI	–0x15	Function code of rel_wai
TFN_IREL_WAI	–0x73	Function code of irel_wai
TFN_SUS_TSK	–0x16	Function code of sus_tsk
TFN_RSM_TSK	–0x17	Function code of rsm_tsk
TFN_FRSM_TSK	–0x18	Function code of frsm_tsk
TFN_DLY_TSK	–0x19	Function code of dly_tsk

[Standard Profile]

The Standard Profile requires support for task dependent synchronization functions.

The Standard Profile requires support for a wakeup request count of one or more. It also requires support for the SUSPENDED state for a task. Therefore, each of

TMAX_WUPCNT and TMAX_SUSCNT must be at least 1.

[Supplemental Information]

TMAX_WUPCNT is undefined if the sleeping state for a task is not supported and is 0 if the wakeup request queuing is not supported. TMAX_SUSCNT is undefined if the SUSPENDED state for a task is not supported, thus, TMAX_SUSCNT is never 0.

[Differences from the μITRON3.0 Specification]

The functions for releasing a task from waiting, rel_wai, and for delaying the invoking task's execution, dly_tsk, are now classified as task dependent synchronization functions.

slp_tsk Put Task to Sleep [S]
tslp_tsk Put Task to Sleep (with Timeout) [S]

[C Language API]

```
ER ercd = slp_tsk ( ) ;
ER ercd = tslp_tsk ( TMO tmout ) ;
```

[Parameter]

TMO	tmout	Specified timeout (only for tslp_tsk)

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_PAR	Parameter error (tmout is invalid; only for tslp_tsk)
E_RLWAI	Forced release from waiting (accept rel_wai while waiting)
E_TMOUT	Polling failure or timeout (only tslp_tsk)

[Functional Description]

These service calls move the invoking task to the sleeping state. However, if wakeup requests are queued, that is, if the wakeup request count for the invoking task is 1 or more, the count is decremented by 1 and the invoking task continues execution.

tslp_tsk has the same functionality as slp_tsk with an additional timeout feature. tmout can be set to a positive number indicating a timeout duration or it can be set to TMO_POL (= 0) or TMO_FEVR (= –1).

[Supplemental Information]

These service calls do not move the invoking task to the WAITING state if wakeup requests for the invoking task are queued. Thus the precedence of the invoking task is not changed.

No polling service call is provided for slp_tsk. If a similar feature is necessary, it can be implemented using can_wup.

wup_tsk Wakeup Task [S]
iwup_tsk [S]

[C Language API]

```
ER ercd = wup_tsk ( ID tskid ) ;
ER ercd = iwup_tsk ( ID tskid ) ;
```

[Parameter]

ID	tskid	ID number of the task to be woken up

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (tskid is invalid or unusable)
E_NOEXS	Non-existent object (specified task is not registered)
E_OBJ	Object state error (specified task is in the DORMANT state)
E_QOVR	Queue overflow (overflow of wakeup request count)

[Functional Description]

These service calls wake up the task specified by tskid from sleeping. The service call that placed the task in the WAITING state will return E_OK to the task.

If the task is not in the sleeping state, the wakeup request for the task is queued. (However, if the task is in the NON-EXISTENT state, an E_NOEXS error is returned and if the task is in the DORMANT state, an E_OBJ error is returned.) Specifically, the wakeup request count is incremented by 1. If the count then exceeds the maximum possible count, an E_QOVR error is returned.

If this service call is invoked from non-task contexts and has its execution delayed, an E_OBJ error and an E_QOVR error may not be returned.

If tskid is TSK_SELF (= 0), the invoking task is specified. If TSK_SELF is specified when this service call is invoked from non-task contexts, an E_ID error is returned.

[Supplemental Information]

The Standard Profile requires the maximum wakeup request count to be at least 1. This implies that a kernel that is compatible with the Standard Profile may not always return an E_QOVR error even if these service calls are invoked on a task with queued wakeup requests.

[Differences from the µITRON3.0 Specification]

The invoking task can now be specified in this service call for the consistency with act_tsk.

can_wup Cancel Task Wakeup Requests [S]

[C Language API]

```
ER_UINT wupcnt = can_wup ( ID tskid ) ;
```

[Parameter]

ID	tskid	ID number of the task for cancelling wakeup requests

[Return Parameter]

ER_UINT	wupcnt	Wakeup request count (positive value or 0) or error code

[Error Code]

E_ID	Invalid ID number (tskid is invalid or unusable)
E_NOEXS	Non-existent object (specified task is not registered)
E_OBJ	Object state error (specified task is in the DORMANT state)

[Functional Description]

This service call cancels all queued wakeup requests for the task specified by tskid and returns the cancelled request count for the task. Specifically, the wakeup request count for the task is cleared to 0. The value returned is the count before it was cleared.

If tskid is TSK_SELF (= 0), the invoking task is specified.

[Supplemental Information]

This service call can be used to check if a task completes a process within a cycle correctly when the task is woken up cyclically. Specifically, can_wup should be invoked when the task completes the process. A return value of 1 or more from can_wup indicates that the next wakeup request is done before the task completes the process in the previous cycle. The task can take measures for this case.

[Differences from the µITRON3.0 Specification]

The wakeup request count (wupcnt) is now the return value of this service call.

rel_wai Release Task from Waiting [S]
irel_wai [S]

[C Language API]

```
ER ercd = rel_wai ( ID tskid ) ;
ER ercd = irel_wai ( ID tskid ) ;
```

[Parameter]

ID	tskid	ID number of the task to be released from waiting

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (tskid is invalid or unusable)
E_NOEXS	Non-existent object (specified task is not registered)
E_OBJ	Object state error (specified task is in the DORMANT state)

[Functional Description]

These service calls forcibly release the task specified by tskid from waiting. Specifically, if the task is in the WAITING state, it is moved to the READY state. If the task is in the WAITNG-SUSPENDED state, it is moved to the SUSPEDED state. When the task is released from waiting by these service calls, the service call that placed the task in the WAITING state will return an E_RLWAI error to the task.

If the task is not in the WAITING state, including the WAITING-SUSPENDED state, an E_OBJ error is returned. However, if the task is in the NON-EXISTENT state, an E_NOEXS error code is returned. If this service call is invoked from non-task contexts and has its execution delayed, an E_OBJ error may not be returned.

[Supplemental Information]

A task cannot specify its own task ID in tskid. If a task does specify its own task ID, these service calls return an E_OBJ error because the task is not in the WAITING state.

These service calls do not cause a task in the SUSPENDED state to resume. frsm_tsk (or rsm_tsk) should be used to forcibly resume a suspended task.

If the task is waiting in a wait queue, the task is removed from the queue. In this case, some other tasks that are in the wait queue may need to be released from waiting. See the functional descriptions of snd_mbf and get_mpl.

The following describes the differences between rel_wai and wup_tsk:

- rel_wai releases a task from any waiting state, while wup_tsk only releases a task from the sleeping state.
- To the task in the sleeping state, a success (E_OK) is returned when the task is

released from sleeping with slp_tsk, while an error (E_RLWAI) is returned when the task is forcibly released from waiting with rel_wai.

- wup_tsk will increment the wakeup request count if the task is not in the sleeping state. On the other hand, rel_wai will return an E_OBJ error if the task is not waiting.

sus_tsk Suspend Task [S]

[C Language API]

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

[Parameter]

ID	tskid	ID number of the task to be suspended

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (tskid is invalid or unusable)
E_CTX	Context error (the invoking task is specified while under dispatching disabled state; any other context error)
E_NOEXS	Non-existent object (specified task is not registered)
E_OBJ	Object state error (specified task is in the DORMANT state)
E_QOVR	Queue overflow (overflow of suspension count)

[Functional Description]

This service call suspends the task specified by tskid. Specifically, if the task is runnable, it is moved to the SUSPENDED state. If the task is in the WAITING state, it is moved to the WAITING-SUSPENDED state. In addition, the suspension count is incremented by 1. If the count then exceeds the maximum possible count, an E_QOVR error is returned.

This service call can be invoked under the dispatching disabled state. However, under the dispatching disabled state, if this service call is invoked specifying the invoking task, an E_CTX error is returned.

If tskid is TSK_SELF (= 0), the invoking task is specified.

[Supplemental Information]

This service call may be invoked under the dispatching disabled state even though the invoking task may be moved to the SUSPENDED state as specified in the parameter. Therefore this is an exception to the principle stating that “The restriction that behavior is undefined when service calls that can move the invoking task to the blocked state are invoked while in the dispatching disabled state applies to a service as a whole.”

The Standard Profile requires the maximum suspension count to be at least 1. This implies that a kernel that is compatible with the Standard Profile may not always return an E_QOVR error even if this service call is invoked on a task in the SUSPENDED state.

[Differences from the μITRON3.0 Specification]

The invoking task can now be specified in tskid.

rsm_tsk	Resume Suspended Task	[S]
frsm_tsk	Forcibly Resume Suspended Task	[S]

[C Language API]

ER ercd = rsm_tsk (ID tskid) ;
ER ercd = frsm_tsk (ID tskid) ;

[Parameter]

ID	tskid	ID number of the task to be resumed

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (tskid is invalid or unusable)
E_NOEXS	Non-existent object (specified task is not registered)
E_OBJ	Object state error (specified task is neither in the SUSPENDED state nor WAITING-SUSPENDED state)

[Functional Description]

These service calls release the task specified by tskid from the SUSPENDED state and allows the task to continue its normal processing. Specifically, the following actions are performed.

rsm_tsk decrements the suspension count of the task by 1. If the count becomes 0, the task is moved according to the following: if the task is in the SUSPEDED state, it is moved to the READY state; if the task is in the WAITING-SUSPENDED state, it is moved to the WAITING state. If the count remains to be 1 or more, the state of the task is not changed.

frsm_tsk clears the suspension count to 0 and forcibly moves the task according to the following: if the task is in the SUSPEDED state, it is moved to the READY state; if the task is in the WAITING-SUSPENDED state, it is moved to the WAITING state.

If the specified task is neither in the SUSPENDED state nor WAITING-SUSPENDED state, an E_OBJ error is returned. However, if the task is in the NON-EXISTING state, an E_NOEXS error is returned.

[Supplemental Information]

A task cannot specify its own task ID in tskid. If a task does specify its own task ID, these service calls return an E_OBJ error because the task is not in the SUSPENDED state. When an implementation-specific service call is capable of moving a task to the SUSPENDED state from non-task contexts or moving the invoking task to the SUSPENDED state under the dispatching disabled state, the invoking task may have the suspension count of 1 or more. The behavior of rsm_tsk and frsm_tsk in this case is implementation-dependent.

[Differences from the μITRON3.0 Specification]

After a task is moved from the SUSPENDED state to the READY state, the task has the lowest precedence among all tasks with the same priority in the READY state. See Section 3.2.1 for more details.

dly_tsk Delay Task [S]

[C Language API]

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

[Parameter]

RELTIM	dlytim	Amount of time to delay the invoking task (relative time)

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_PAR	Parameter error (dlytim is invalid)
E_RLWAI	Forced release from waiting (accept rel_wai while waiting)

[Functional Description]

This service call delays the execution of the invoking task for the amount of time specified in dlytim. Specifically, the invoking task is set to be released from waiting when the specified relative time has passed since the invocation of this service call, and then it is moved to the delayed state. When the task is released from waiting after the relative time expires, the service call completes and returns E_OK.

dlytim is the relative time when the task is released from the delayed state with respect to the time when the service call is invoked.

[Supplemental Information]

The release of a task from the delayed state depends on the system time. Therefore, the task is released at the first time tick after the specified time has passed. The system must guarantee that the release of the task occurs after an elapsed time equal to or greater than the specified time (see Section 2.1.9). This service call moves the invoking task to the delayed state even if dlytim is 0.

The delayed state is a kind of the WAITING state and can be forcibly released with rel_wai. The delayed time includes the time a task spends in the WAITING-SUSPENDED state.

This service call is different from tslp_tsk in that it returns E_OK when the specified time expires. Also, an invocation of wup_tsk for the task will not release the task from the delayed state. Only ter_tsk and rel_wai can release the task from the delayed state before the time expires.

[Differences from the µITRON3.0 Specification]

The data type of dlytim has been changed from DLYTIME to RELTIM.

4.3 Task Exception Handling Functions

Task exception handling functions provide handling task exceptions within the task's context. Task exception handling functions include the ability to define a task exception handling routine, to request a task exception handling, to enable and disable task exception handling, and to reference the state of a task exception handling.

When a task's exception handling is requested, the task suspends processing and the task exception handling routine is started. The handling routine runs within the same context as the task itself. Once the task exception handling routine returns, the task resumes processing. An application can register a task exception handling routine for each task. A task exception handling routine is not registered when the task is created.

When a task exception handling is requested, the task exception code representing the type of exception is specified. For each task, the kernel manages the exception code representing the exceptions that have been requested but have not been processed yet. This code is referred to as the pending exception code. The pending exception code is 0 if no unprocessed exception request exists. When a task exception handling is requested for a task that has unprocessed exception requests, the task's pending exception code is bit-wise ORed with the requested exception code. The pending exception code is cleared to 0 when the task is activated.

A task can be in either the task exception disabled state or the task exception enabled state. Moving a task to the task exception disabled state is called "disabling task exceptions." Moving a task to the exception enabled state is called "enabling task exceptions." Just after a task starts, it is in the task exception disabled state.

The following behavior is implementation-defined. The kernel disables task exceptions when an extended service routine is started and restores the original state when the routine returns. In addition, if ena_tex is invoked from an extended service call routine, an E_CTX error is returned because task exceptions should be kept disabled during the execution of the routine.

A task's exception handling routine is started when the following four conditions are met: task exceptions are enabled for the task, the task's pending exception code is not 0, the task is in the RUNNING state, and non-task contexts or CPU exception handlers are not being executed. The pending exception code (texptn) and the task's extended information (exinf) are passed to the task exception handling routine as parameters. At this point, task exceptions are disabled and the pending exception code is cleared to 0.

When the task exception handling routine returns, the task resumes executing the process that was executing before the routine was started. At this point, the task exceptions are enabled. If the pending exception code is not 0, the task exception handling routine is restarted.

The following data type is used for task exception handling functions:

TEXPTN Bit pattern for the task exception code (unsigned integer)

The format to write a task exception handling routine in C language is shown below:

```
void texrtn ( TEXPTN texptn, VP_INT exinf )
{
        /* Body of the task exception handling routine */
}
```

The following kernel configuration constant is defined for use with task exception handling functions:

TBIT_TEXPTN The number of bits in the task exception code (the number of bits of TEXPTN type)

The following packet data types are defined for defining and referencing task exception handling routines:

```
typedef struct t_dtex {
        ATR     texatr ;    /* Task exception handling routine
                               attribute */
        FP      texrtn ;    /* Task exception handling routine start
                               address */
        /* Other implementation specific fields may be added. */
} T_DTEX ;

typedef struct t_rtex {
        STAT    texstat ;   /* Task exception state */
        TEXPTN  pndptn ;    /* Pending exception code */
        /* Other implementation specific fields may be added. */
} T_RTEX ;
```

The following represents the function codes for the task exception handling service calls:

TFN_DEF_TEX	–0x1b	Function code of def_tex
TFN_RAS_TEX	–0x1c	Function code of ras_tex
TFN_IRAS_TEX	–0x74	Function code of iras_tex
TFN_DIS_TEX	–0x1d	Function code of dis_tex
TFN_ENA_TEX	–0x1e	Function code of ena_tex
TFN_SNS_TEX	–0x1f	Function code of sns_tex
TFN_REF_TEX	–0x20	Function code of ref_tex

[Standard Profile]

The Standard Profile requires support for task exception handling functions except for dynamic definition of an exception handling routine (def_tex) and reference of a task exception handling routine state (ref_tex).

The Standard Profile also requires the bit-width for the bit pattern data type to be at least 16 bits:

TEXPTN 16 bits or more

Therefore, TBIT_TEXPTN must be 16 or more.

[Supplemental Information]

The specification does not specify whether a task exception handling routine is started in the CPU locked state because the behavior of service calls that request task exception handling in the CPU locked state is undefined. On the other hand, a task exception handling routine must be started if the four task exception handling conditions are met and even if dispatching is disabled.

The context and states under which task exception handling routines execute are summarized below:

- Task exception handling routines execute in the same context as the tasks (see Section 3.5.1). The contexts in which the task exception handling routines execute are classified as task contexts.
- The start of and the return from the task exception handling routines do not change the CPU state or the dispatching state (see Sections 3.5.4 and 3.5.5). However, the specification does not specify whether a task exception handling routine is started in the CPU locked state.

The circumstances regarding enabling and disabling task exceptions are summarized below:

- When a task is activated, task exceptions for the task are disabled.
- When a task exception handling routine is started, task exceptions are disabled. Task exceptions are enabled upon the return from the task exception handling routine.
- Invoking dis_tex disables task exceptions and invoking ena_tex enables task exceptions.
- When the definition of a task exception handling routine is released with def_tex, task exceptions are disabled.

Task exception handling routines may execute a non-local jump by invoking longjmp from the standard C library. This is allowed because the exception handling routine executes within the context of the task. When a non-local jump is used to terminate a task exception handling routine, the kernel does not enable task exceptions because the kernel cannot detect whether the task exception handling routine terminates. The application may enable the task exceptions by invoking ena_tex. In addition if an application executes a non-local jump from the task exception handling routine, the application must disable task exceptions in order to maintain integrity of global data structures (see Rationale below).

A task exception handling routine may be restarted just after it returns. In this case, the stack pointer must be the same as the stack pointer when the routine is started previously. This implies there is no wasted stack space when the task exception handling routine is restarted after its completion. If this were not the case, it would be impossible to bound the size of the stack area used by a succession of task exception handling routines.

The μITRON4.0 Specification does not provide the functionality to mask a task's exception code bit by bit. However, an application could still realize this functionality through the specified task exception handling functions as described below. An application manages the task exception handling mask for each task. At the beginning of the task exception handling routine, the application checks if the passed task exception code is masked or not. If the code is masked, the routine must record that the routine was started with the exception code and return immediately. To be accurate, the routine must handle the case where some part of the code is masked and some part of the code is not masked. Later, when the application changes the task exception handling mask, the application must check if the exception handling routine was started with a previously masked exception code. If there is a record of a masked exception, the task exception handling routine is started by the application to handle the exception.

Task exception handling routines are not nested because task exceptions are disabled at the start of the task exception handling routine. If task exception handling routines are complex, especially when the task can enter the WAITING state, there are cases when the routine may need to be nested because an exception could occur while the exception routine is executing. In cases like this, the exception routine can be nested by invoking `ena_tex` within the task exception handling routine. However, some measures must be taken to avoid starting an unlimited number of nested task exception handling routines. An example measure is to mask the currently processing exceptions with the exception handling mask described above.

If a CPU exception occurs while a task exception handling routine is executing, the CPU exception handler begins executing. Once the CPU exception handler returns, the task exception handler resumes even if the CPU exception handler requests task exception handling. This is because the task exceptions were disabled when the task exception handling routine started. If the cause of the CPU exception is not removed within the CPU exception handler, the CPU exception is raised again just after the handler returns. As a result, the CPU exception will continue forever. This also applies to any CPU exceptions that occurred while in the task exception disabled state.

In principle the application must avoid cases where a CPU exception occurs while in the task exception disabled state, when the CPU exception handler requests task exception handling. However, CPU exceptions may not be avoidable due to software and/or hardware malfunctions. In order to avoid continuous CPU exceptions where CPU exceptions are unavoidable, the CPU exception handler must reference the task exception handling state and perform special handling when task exceptions are disabled. Nesting the execution of task exception handling routines using the previously described method, may also be necessary to shorten the duration task exceptions are disabled.

In an implementation where different stack spaces are used for the application and the kernel, information stored in the kernel stack space or in the task control block (TCB) must often be moved to the application stack space in order to support the nesting of task exception handling routines. For instance, if a task exception handling request occurs while a task is being preempted, the exception routine will start the next time when the task enters the READY state. In this case, the task's states before the preemption, which is stored in the kernel stack space or in the TCB, must be moved to the application's stack space. When the task exception handling routine returns, the task states must be restored based on the information stored in the application's stack space.

[Differences from the μITRON3.0 Specification]

Task exception handling functions have been newly added.

[Rationale]

The μITRON4.0 Specification only includes basic task exception handling functions. An application can realize more complex exception handling based on the provided functions when necessary. This allows the application to gain more powerful support while keeping the kernel compact.

The specification only states that task exception handling routines execute in the same context as the task. The description regarding non-local jumps via `longjmp` is included in the supplemental information because easy use of `longjmp` from the task exception handling routine is dangerous for the reason described in the next paragraph. A task exception handling routine can safely be terminated forcibly through `ext_tsk`. This method is considered to be sufficient for the scope of the Standard Profile.

An easy use of `longjmp` can result to the following. If a task exception handling routine is started while a global data structure is being operated on and if the task exits the task exception handling routine with a `longjmp`, the possibility exists that the data is corrupted. In such cases, users should be very careful when using of `longjmp` to exit the task exception handling routine. Specifically, task exceptions must be disabled while a global data structure is inconsistent.

DEF_TEX Define Task Exception Handling Routine (Static API) [S]
def_tex Define Task Exception Handling Routine

[Static API]

DEF_TEX (ID tskid, { ATR texatr, FP texrtn }) ;

[C Language API]

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

[Parameter]

ID	tskid	ID number of the task to be defined
T_DTEX *	pk_dtex	Pointer to the packet containing the task exception handling routine definition information (in DEF_TEX, the contents must be directly specified.)

pk_dtex includes (T_DTEX type)

ATR	texatr	Task exception handling routine attribute
FP	texrtn	Task exception handling routine start address

(Other implementation specific information may be added.)

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (tskid is invalid or unusable)
E_NOEXS	Non-existent object (specified task is not registered)
E_RSATR	Reserved attribute (texatr is invalid or unusable)
E_PAR	Parameter error (pk_dtex or texrtn is invalid)

[Functional Description]

This service call defines the task exception handling routine for the task specified by tskid based on the information contained in the packet pointed to by pk_dtex. texatr is the attribute of the task exception handling routine. texrtn is the start address of the task exception handling routine.

In DEF_TEX, tskid is an integer parameter without automatic assignment. texatr is a preprocessor constant expression parameter.

If pk_dtex is NULL (= 0), the task exception handling routine currently defined is released and the task exception handling routine becomes undefined. At this time, the pending exception code is cleared to 0 and task exception are disabled. When a new task exception handling routine is defined over top of an old one, the old one is released and the new one takes its place. Under this condition, the pending exception code is not cleared and task exceptions are not disabled.

When tskid is TSK_SELF (= 0), the task exception handling routine is defined for the invoking task.

texatr can be specified as (TA_HLNG || TA_ASM). If TA_HLNG (= 0x00) is specified, the task exception handling routine is started through the C language interface. If TA_ASM (= 0x01) is specified, the routine is started through an assembly language interface.

[Standard Profile]

The Standard Profile does not require support for when TA_ASM is specified in texatr.

[Supplemental Information]

The task exception handling routine remains effective until def_tex is invoked with pk_dtex set to NULL or until the task is deleted.

When DEF_TEX is used to define a task exception handling routine for a task, the task must be created with CRE_TSK appearing before DEF_TEX in the system configuration file.

[Rationale]

When the definition of the task exception handling routine is cancelled, the pending exception code is cleared and task exceptions are disabled. This is done to keep the pending exception code to 0 and task exceptions to the disabled state, when the task exception handling routine is not defined. Once a task exception handling routine becomes undefined, these conditions are kept because the pending exception code cannot be set and because task exceptions cannot be enabled.

ras_tex Raise Task Exception Handling [S]
iras_tex [S]

[C Language API]

ER ercd = ras_tex (ID tskid, TEXPTN rasptn) ;
ER ercd = iras_tex (ID tskid, TEXPTN rasptn) ;

[Parameter]

ID	tskid	ID number of the task requested
TEXPTN	rasptn	Task exception code to be requested

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (tskid is invalid or unusable)
E_NOEXS	Non-existent object (specified task is not registered)
E_PAR	Parameter error (rasptn is invalid)
E_OBJ	Object state error (specified task is in the DORMANT state, task exception handling routine is not defined)

[Functional Description]

These service calls request task exception handling for the task specified by tskid. The task exception code is specified by the bit pattern in rasptn. Specifically, the task's pending exception code is bit-wise ORed with the requested exception code.

If tskid is TSK_SELF (= 0), the invoking task is specified. If TSK_SELF is specified when this service call is invoked from non-task contexts, an E_ID error is returned.

If the task is in the DORMANT state or if the task exception handling routine for the task is undefined, an E_OBJ error is returned. If the service call is invoked from non-task contexts and has its execution delayed, an E_OBJ error may not be returned.

If rasptn is 0, an E_PAR error is returned.

[Supplemental Information]

These service calls start the task exception handling routine if all the conditions for starting the routine are met.

If the task is in the blocked state, these service calls only update the pending exception code, and do not release the task from waiting nor from the SUSPENDED state. If the task must be released from the block state, the application can use rel_wai or frsm_tsk (or rsm_tsk) to do so.

There are many service calls that when invoked from non-task contexts can have their execution delayed until the system is in a state where dispatching can occur. However, this service call must be executed even if the system is in the dispatching disabled state. For example, if an interrupt handler requests a task exception handling for the task in the RUNNING state while in the dispatching disabled state, the task exception handling routine must be started just after the return from the interrupt handler. This is useful for stopping a malfunctioning task running with dispatching disabled by requesting a task exception handling from an interrupt handler. However, this is not useful for stopping a task running with the CPU locked or a task running with task exceptions and dispatching disabled.

dis_tex Disable Task Exceptions [S]

[C Language API]

```
ER ercd = dis_tex ( ) ;
```

[Parameter]

None

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_OBJ	Object state error (task exception handling routine is not defined)

[Functional Description]

This service call moves the invoking task to the task exception disabled state. If the task exception handling routine is not defined for the invoking task, an E_OBJ error is returned.

ena_tex Enable Task Exceptions [S]

[C Language API]

ER ercd = ena_tex ( ) ;

[Parameter]

None

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_OBJ	Object state error (the task exception handling routine is not defined)
E_CTX	Context error (invoked from a context not capable of enabling task exceptions, any other context errors)

[Functional Description]

This service call moves the invoking task to the task exception enabled state. If the task exception handling routine is not defined for the invoking task, an E_OBJ error is returned.

For an implementation that does not allow task exceptions enabled within an extended service call routine, an E_CTX error is returned if this service call is invoked from an extended service call routine.

[Supplemental Information]

This service call starts the task exception handling routine if all the conditions for starting the routine are met.

sns_tex Reference Task Exception Handling State [S]

[C Language API]

BOOL state = sns_tex ( ) ;

[Parameter]

None

[Return Parameter]

BOOL	state	Task exception disabled state

[Functional Description]

This service call returns TRUE if task exceptions are disabled for the task in the RUNNING state (which corresponds to the invoking task when this service call invoked from task contexts) and returns FALSE if task exceptions are enabled. If this service call is invoked from non-task contexts and there is no task in the RUNNING state, TRUE is returned.

[Supplemental Information]

Tasks that have no defined task exception handling routine always have task exceptions disabled. Therefore, when the invoking task has no defined task exception handling routine, this service call returns TRUE.

ref_tex Reference Task Exception Handling State

[C Language API]

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

[Parameter]

ID	tskid	ID number of the task to be referenced
T_RTEX *	pk_rtex	Pointer to the packet returning the task exception handling state

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

pk_rtex includes (T_RTEX type)

STAT	texstat	Task exception handling state
TEXPTN	pndptn	Pending exception code

(Other implementation specific information may be added.)

[Error Code]

E_ID	Invalid ID number (tskid is invalid or unusable)
E_NOEXS	Non-existent object (specified task is not registered)
E_PAR	Parameter error (pk_rtex is invalid)
E_OBJ	Object state error (specified task is in the DORMANT state, the task exception handling routine is not defined)

[Functional Description]

This service call references the state of the task exception handling for the task specified by tskid. The state of the task exception handling is returned through the packet pointed to by pk_rtex.

texstat can take on any of the following values:

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

The pending exception code is returned through pndptn. If no unprocessed exception request exists, pndptn is 0.

If tskid is TSK_SELF (= 0), the state of the invoking task is referenced.

If the task is in the DORMANT state or the task exception handling routine is not defined for the task, an E_OBJ error is returned.

4.4 Synchronization and Communication Functions

Synchronization and communication functions provide synchronization and communication between tasks through objects that are independent of the tasks. The objects are semaphores, data queues, event flags, and mailboxes.

[Differences from the μITRON3.0 Specification]

Implementation of mailboxes are now limited to linked lists. Data queues have been newly introduced and provide the same functionality as mailboxes but are implemented with ring buffers.

4.4.1 Semaphores

A semaphore is an object used for mutual exclusion and synchronization. A semaphore indicates availability and the number of unused resources by a resource count. Semaphore functions include the ability to create and delete a semaphore, to acquire and release resources, and to reference the state of a semaphore. A semaphore is an object identified by an ID number. The ID number of a semaphore is called the semaphore ID.

A semaphore has an associated resource count and a wait queue. The resource count indicates the resource availability or the number of unused resources. The wait queue manages the tasks waiting for resources from the semaphore. When a task releases a semaphore resource, the resource count is incremented by 1. When a task acquires a semaphore resource, the resource count is decremented by 1. If a semaphore has no resources available or more precisely the resource count is 0, a task attempting to acquire a resource will wait in the wait queue until a resource is returned to the semaphore.

In order to avoid the case where too many resources are returned to a semaphore, each semaphore has a maximum resource count indicating the maximum number of unused resources available to the semaphore. If more resources are returned to the semaphore than its maximum resource count, an error will be returned.

The following kernel configuration constant is defined for use with semaphore functions:

TMAX_MAXSEM	Maximum value of the maximum definable semaphore resource count

The following data type packets are defined for creating and referencing semaphores:

```
typedef struct t_csem {
        ATR     sematr ;    /* Semaphore attribute */
        UINT    isemcnt ;   /* Initial semaphore resource count */
        UINT    maxsem ;    /* Maximum semaphore resource count */
        /* Other implementation specific fields may be added. */
```

```
} T_CSEM ;

typedef struct t_rsem {
        ID          wtskid ;    /* ID number of the task at the head of the
                                   semaphore's wait queue */
        UINT        semcnt ;    /* Current semaphore resource count */
        /* Other implementation specific fields may be added. */
} T_RSEM ;
```

The following represents the function codes for the semaphore service calls:

TFN_CRE_SEM	–0x21	Function code of cre_sem
TFN_ACRE_SEM	–0xc2	Function code of acre_sem
TFN_DEL_SEM	–0x22	Function code of del_sem
TFN_SIG_SEM	–0x23	Function code of sig_sem
TFN_ISIG_SEM	–0x75	Function code of isig_sem
TFN_WAI_SEM	–0x25	Function code of wai_sem
TFN_POL_SEM	–0x26	Function code of pol_sem
TFN_TWAI_SEM	–0x27	Function code of twai_sem
TFN_REF_SEM	–0x28	Function code of ref_sem

[Standard Profile]

The Standard Profile requires support for semaphore functions except for dynamic creation and deletion of a semaphore (cre_sem, acre_sem, del_sem) and reference of a semaphore state (ref_sem).

The Standard Profile requires that maximum resource count can be set to at least 65535. Although TMAX_MAXSEM does not have to be defined, if it is defined, it must be equal to or greater than 65535.

[Rationale]

TMAX_MAXSEM is only used when semaphores are dynamically created. Since dynamic semaphore creation does not have to be supported in the Standard Profile, TMAX_MAXSEM does not have to be defined in this case.

CRE_SEM	Create Semaphore (Static API)	[S]
cre_sem	Create Semaphore	
acre_sem	Create Semaphore (ID Number Automatic Assignment)	

[Static API]

```
CRE_SEM ( ID semid, { ATR sematr, UINT isemcnt,
                       UINT maxsem } ) ;
```

[C Language API]

```
ER ercd = cre_sem ( ID semid, T_CSEM *pk_csem ) ;
ER_ID semid = acre_sem ( T_CSEM *pk_csem ) ;
```

[Parameter]

ID	semid	ID number of the semaphore to be created (except acre_sem)
T_CSEM *	pk_csem	Pointer to the packet containing the semaphore creation information (in CRE_SEM, packet contents must be directly specified.)

pk_csem includes (T_CSEM type)

ATR	sematr	Semaphore attribute
UINT	isemcnt	Initial semaphore resource count
UINT	maxsem	Maximum semaphore resource count

(Other implementation specific information may be added.)

[Return Parameter]

cre_sem:

ER	ercd	E_OK for normal completion or error code

acre_sem:

ER_ID	semid	ID number (positive value) of the created semaphore or error code

[Error Code]

E_ID	Invalid ID number (semid is invalid or unusable; only cre_sem)
E_NOID	No ID number available (there is no semaphore ID assignable; only acre_sem)
E_RSATR	Reserved attribute (sematr is invalid or unusable)
E_PAR	Parameter error (pk_csem, isemcnt, or maxsem is invalid)
E_OBJ	Object state error (specified semaphore is already registered; only cre_sem)

[Functional Description]

These service calls create a semaphore with an ID number specified by semid based on the information contained in the packet pointed to by pk_csem. sematr is the attribute of the semaphore. isemcnt is the initial value of the resource count after creation of the semaphore. maxsem is the maximum resource count of the semaphore.

In CRE_SEM, semid is an integer parameter with automatic assignment. sematr is a preprocessor constant expression parameter.

acre_sem assigns a semaphore ID from the pool of unassigned semaphore IDs and returns the assigned semaphore ID.

sematr can be specified as (TA_FIFO || TA_TPRI). If TA_FIFO (= 0x00) is specified, the semaphore's wait queue will be in FIFO order. If TA_TPRI (= 0x01) is specified, the semaphore's wait queue will be in task priority order.

[Differences from the μITRON3.0 Specification]

The extended information has been removed from the semaphore creation information. The data types of isemcnt and maxsem have been changed from INT to UINT.

acre_sem has been newly added.

del_sem Delete Semaphore

[C Language API]

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

[Parameter]

ID	semid	ID number of the semaphore to be deleted

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (semid is invalid or unusable)
E_NOEXS	Non-existent object (specified semaphore is not registered)

[Functional Description]

This service call deletes the semaphore specified by semid.

[Supplemental Information]

See Section 3.8 for information regarding handling tasks that are waiting for a resource in a semaphore's wait queue when the semaphore is deleted.

sig_sem Release Semaphore Resource [S]
isig_sem [S]

[C Language API]

```
ER ercd = sig_sem ( ID semid ) ;
ER ercd = isig_sem ( ID semid ) ;
```

[Parameter]

ID	semid	ID number of the semaphore to which resource is released

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (semid is invalid or unusable)
E_NOEXS	Non-existent object (specified semaphore is not registered)
E_QOVR	Queue overflow (release will exceed maximum resource count)

[Functional Description]

These service calls release one resource to the semaphore specified by semid. If any tasks are waiting for the specified semaphore, the task at the head of the semaphore's wait queue is released from waiting. When this happens, the associated semaphore resource count is not changed. The released task receives E_OK from the service call that caused it to wait in the semaphore's wait queue. If no tasks are waiting for the specified semaphore, the semaphore resource count is incremented by 1.

These service calls return an E_QOVR error if incrementing the resource count by 1 will cause the count to exceed the maximum semaphore resource count. If this service call is invoked from non-task contexts and has its execution delayed, an E_QOVR error may not be returned, however the condition must still be checked.

wai_sem	Acquire Semaphore Resource	[S]
pol_sem	Acquire Semaphore Resource (Polling)	[S]
twai_sem	Acquire Semaphore Resource (with Timeout)	[S]

[C Language API]

```
ER ercd = wai_sem ( ID semid ) ;
ER ercd = pol_sem ( ID semid ) ;
ER ercd = twai_sem ( ID semid, TMO tmout ) ;
```

[Parameter]

ID	semid	ID number of the semaphore from which resource is acquired
TMO	tmout	Specified timeout (only twai_sem)

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (semid is invalid or unusable)
E_NOEXS	Non-existent object (specified semaphore is not registered)
E_PAR	Parameter error (tmout is invalid; only twai_sem)
E_RLWAI	Forced release from waiting (accept rel_wai while waiting; except pol_sem)
E_TMOUT	Polling failure or timeout (except wai_sem)
E_DLT	Waiting object deleted (semaphore is deleted while waiting; except pol_sem)

[Functional Description]

There service calls acquire one resource from the semaphore specified by semid. If the resource count of the specified semaphore is 1 or more, the associated resource count is decremented by 1. In this case, the invoking task is not moved to the WAITING state, but rather receives a normal return from the service call. If, on the other hand, the resource count of the specified semaphore is 0, the invoking task is placed in the semaphore's wait queue and is moved to the waiting state for the semaphore. In this case, the resource count remains unchanged at 0.

If there are already tasks in the wait queue, the invoking task is placed in the wait queue as described below. When the semaphore's attribute has TA_TFIFO (= 0x00) set, the invoking task is placed in the tail of the wait queue. When the attribute has TA_TPRI (= 0x01) set, the invoking task is placed in the wait queue in the order of the task's priority. If the wait queue contains tasks with the same priority as the invoking task, the invoking task is placed after those tasks.

pol_sem is a polling service call with the same functionality as wai_sem.

twai_sem has the same functionality as wai_sem with an additional timeout feature. tmout can be set to a positive number indicating a timeout duration or it can be set to TMO_POL (= 0) or TMO_FEVR (= –1).

[Supplemental Information]

twai_sem acts the same as pol_sem if TMO_POL is specified in tmout as long as no context error occurs. Also, twai_sem acts the same as wai_sem if TMO_FEVR is specified in tmout.

[Differences from the μITRON3.0 Specification]

The name of the polling service call has been changed from preq_sem to pol_sem.

ref_sem Reference Semaphore State

[C Language API]

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

[Parameter]

ID	semid	ID number of the semaphore to be referenced
T_RSEM *	pk_rsem	Pointer to the packet returning the semaphore state

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

pk_rsem includes (T_RSEM type)

ID	wtskid	ID number of the task at the head of the semaphore's wait queue
UINT	semcnt	Current semaphore resource count

(Other implementation specific information may be added.)

[Error Code]

E_ID	Invalid ID number (semid is invalid or unusable)
E_NOEXS	Non-existent object (specified semaphore is not registered)
E_PAR	Parameter error (pk_rsem is invalid)

[Functional Description]

This service call references the state of the semaphore specified by semid. The state of the semaphore is returned through the packet pointed to by pk_rsem.

The ID number of the task at the head of the semaphore's wait queue is returned through wtskid. If no tasks are waiting for the semaphore's resource, TSK_NONE (= 0) is returned instead.

The semaphore's current resource count is returned through semcnt.

[Supplemental Information]

A semaphore cannot have wtskid ≠ TSK_NONE and semcnt ≠ 0 at the same time.

[Differences from the µITRON3.0 Specification]

The extended information has been removed from the reference information. The ID number of the task at the head of the wait queue is now returned instead of a boolean value indicating whether a task is waiting or not. Based on this replacement, the name and data type of the return parameter has been changed.

The data type of semcnt has been changed from INT to UINT. The order of parameters and of return parameters has been changed.

4.4.2 Eventflags

An eventflag is a synchronization object that consists of multiple bits in a bit pattern where each bit represents an event. Eventflag functions include the ability to create and delete an eventflag, to set and clear an eventflag, to wait for an eventflag, and to reference the state of an eventflag. An eventflag is an object identified by an ID number. The ID number of an eventflag is called the eventflag ID.

An eventflag has an associated bit pattern expressing the state of its events, and a wait queue for tasks waiting on these events. Sometimes the bit pattern of an eventflag is simply called an eventflag. A task is able to set specified bits when an event occurs and is able to clear specified bits when necessary. Tasks waiting for events to occur will wait until every specified bit in the eventflag bit pattern is set. Tasks waiting for an eventflag are placed in the eventflag's wait queue.

The following data type is used for eventflag functions:

FLGPTN	Bit pattern of the eventflag (unsigned integer)

The following kernel configuration constant is defined for use with eventflag functions:

TBIT_FLGPTN	The number of bits in an eventflag

The following kernel configuration constant is defined for use with eventflag functions:

```
typedef struct t_cflg {
        ATR         flgatr ;        /* Eventflag attribute */
        FLGPTN      iflgptn ;       /* Initial value of the eventflag bit
                                       pattern */
        /* Other implementation specific fields may be added. */
} T_CFLG ;

typedef struct t_rflg {
        ID          wtskid ;        /* ID number of the task at the head of the
                                       eventflag's wait queue */
        FLGPTN      flgptn ;        /* Current eventflag bit pattern */
        /* Other implementation specific fields may be added. */
} T_RFLG ;
```

The following represents the function codes for the eventflag service calls:

TFN_CRE_FLG	–0x29	Function code of cre_flg
TFN_ACRE_FLG	–0xc3	Function code of acre_flg
TFN_DEL_FLG	–0x2a	Function code of del_flg
TFN_SET_FLG	–0x2b	Function code of set_flg
TFN_ISET_FLG	–0x76	Function code of iset_flg
TFN_CLR_FLG	–0x2c	Function code of clr_flg
TFN_WAI_FLG	–0x2d	Function code of wai_flg
TFN_POL_FLG	–0x2e	Function code of pol_flg
TFN_TWAI_FLG	–0x2f	Function code of twai_flg
TFN_REF_FLG	–0x30	Function code of ref_flg

[Standard Profile]

The Standard Profile requires support for eventflag functions except for dynamic creation and deletion of an eventflag (cre_flg, acre_flg, del_flg) and reference of an eventflag state (ref_flg).

The Standard Profile does not require support for multiple tasks waiting for an eventflag, i.e. eventflags with the TA_WMUL attribute.

The Standard Profile requires support for an eventflag's bit pattern of at least 16 bits. Therefore, TBIT_FLGPTN must be defined to be at least 16. The minimum bit width of the data type for eventflag functions is as follows:

FLGPTN	16 bits or more

[Supplemental Information]

There is no limitation to the number of bits supported by an eventflag except when implementing the Standard Profile. Therefore it is possible to supply an eventflag that supports only 1 bit. Because the C language does not support a data type with an arbitrary bit width, the number of bits in a variable of FLGPTN type may actually be more than the number of bits defined in TBIT_FLGPTN (the number of bits in an eventflag).

[Differences from the μITRON3.0 Specification]

The data type of the parameter holding an eventflag bit pattern has been changed from UINT to the new data type FLGPTN.

CRE_FLG	Create Eventflag (Static API)	[S]
cre_flg	Create Eventflag	
acre_flg	Create Eventflag (ID Number Automatic Assignment)	

[Static API]

CRE_FLG (ID flgid, { ATR flgatr, FLGPTN iflgptn }) ;

[C Language API]

ER ercd = cre_flg (ID flgid, T_CFLG *pk_cflg) ;

ER_ID flgid = acre_flg (T_CFLG *pk_cflg) ;

[Parameter]

ID	flgid	ID number of the eventflag to be created (except acre_flg)
T_CFLG *	pk_cflg	Pointer to the packet containing the eventflag creation information (in CRE_FLG, packet contents must be directly specified.)

pk_cflg includes (T_CFLG type)

ATR	flgatr	Eventflag attribute
FLGPTN	iflgptn	Initial value of eventflag bit pattern

(Other implementation specific information may be added.)

[Return Parameter]

cre_flg:

ER	ercd	E_OK for normal completion or error code

acre_flg:

ER_ID	flgid	ID number (positive value) of the created eventflag or error code

[Error Code]

E_ID	Invalid ID number (flgid is invalid or unusable; only cre_flg)
E_NOID	No ID number available (there is no eventflag ID assignable; only acre_flg)
E_RSATR	Reserved attribute (flgatr is invalid or unusable)
E_PAR	Parameter error (pk_cflg or iflgptn is invalid)
E_OBJ	Object state error (specified eventflag is already registered; only cre_flg)

[Functional Description]

These service calls create an eventflag with an ID number specified by flgid based on the information contained in the packet pointed to by pk_cflg. flgatr is the attribute of the eventflag. iflgptn is the initial value of the bit pattern after creation of the eventflag.

In CRE_FLG, flgid is an integer parameter with automatic assignment. flgatr is a preprocessor constant expression parameter.

acre_flg assigns an eventflag ID from the pool of unassigned eventflag IDs and returns the assigned eventflag ID.

flgatr can be specified as ((TA_TFIFO || TA_TPRI) | (TA_WSGL || TA_WMUL) | [TA_CLR]). If TA_TFIFO (= 0x00) is specified, the eventflag's wait queue will be in FIFO order. If TA_TPRI (= 0x01) is specified, the eventflag's wait queue will be in task priority order. If TA_WSGL (= 0x00) is specified, only a single task is allowed to be in the waiting state for the eventflag. If TA_WMUL (= 0x02) is specified, multiple tasks are allowed to be in the waiting state for the eventflag. If TA_CLR (= 0x04) is specified, the eventflag's entire bit pattern will be cleared when a task is released from the waiting state for the eventflag.

[Standard Profile]

The Standard Profile does not require support for when TA_WMUL is specified in flgatr.

[Supplemental Information]

A task in the waiting state for an eventflag is not always released from waiting according to its order in the wait queue. This is because when the task satisfies the release condition, it is released from waiting even if it is not at the head of the wait queue. For example, even if an eventflag's attribute has TA_TFIFO set, tasks are not always released from the wait queue in FIFO order.

If TA_WSGL is specified in flgatr, the eventflag with the TA_TFIFO attribute behaves the same as the eventflag with the TA_TPRI attribute.

Multiple tasks cannot be released from the waiting state for an eventflag with the TA_CLR attribute. This is because when a task is released from waiting, all of the bits in the eventflag is cleared.

[Differences from the μITRON3.0 Specification]

The specification of clearing an eventflag has been moved from the wait mode parameter in wai_flg to the eventflag attribute. This change has been made because there is almost never a case where some waiting tasks will require the bit pattern to be cleared and some tasks will require the bit pattern to remain intact.

The functionality allowing the eventflag's wait queue to be ordered by task priority with the TA_TPRI attribute has been added.

The extended information has been removed from the eventflag creation information. The data type of iflgptn have been changed from the UINT to FLGPTN. The value of TA_WMUL has been changed.

acre_flg has been newly added.

del_flg Delete Eventflag

[C Language API]

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

[Parameter]

ID	flgid	ID number of the eventflag to be deleted

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (flgid is invalid or unusable)
E_NOEXS	Non-existent object (specified eventflag is not registered)

[Functional Description]

This service call deletes the eventflag specified by flgid.

[Supplemental Information]

See Section 3.8 for information regarding handling tasks that are waiting in an eventflag's wait queue when the eventflag is deleted.

set_flg Set Eventflag [S]
iset_flg [S]

[C Language API]

```
ER ercd = set_flg ( ID flgid, FLGPTN setptn ) ;
ER ercd = iset_flg ( ID flgid, FLGPTN setptn ) ;
```

[Parameter]

ID	flgid	ID number of the eventflag to be set
FLGPTN	setptn	Bit pattern to set

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (flgid is invalid or unusable)
E_NOEXS	Non-existent object (specified eventflag is not registered)
E_PAR	Parameter error (setptn is invalid)

[Functional Description]

These service calls set the bits specified by setptn in the eventflag specified by flgid. Specifically, the bit pattern of the eventflag is updated to the bit-wise OR of its bit pattern before the invocation of the service call with the value specified in setptn.

After the eventflag's bit pattern is updated, any tasks that satisfy their release conditions are released from waiting. Specifically, each task in the eventflag's wait queue is checked starting from the head and is released from waiting if its release condition is satisfied. Each of the released tasks receives E_OK from the service call that caused it to wait in the eventflag's wait queue. It also receives the bit pattern of the eventflag satisfying the task's releasing condition. If the eventflag's attribute has TA_CLR (= 0x04) set, the service calls complete after clearing the entire bit pattern of the eventflag. If TA_CLR is not specified, the remaining tasks in the wait queue are checked to see if they satisfy their release conditions. The service calls terminate after all tasks have been checked. See the functional description of wai_flg for information about tasks' release conditions.

Multiple tasks can be released by a single invocation of set_flg if the eventflag's attribute has the TA_WMUL (= 0x02) attribute but not the TA_CLR attribute set. When multiple tasks are released, they are released in the same order as in the eventflag's wait queue. Therefore, among the same priority tasks that are moved to the READY state, a task closer to the head of the wait queue will have higher precedence.

[Supplemental Information]

No action is required when all of the bits of setptn are 0.

[Differences from the μITRON3.0 Specification]

The data type of `setptn` has been changed from UINT to FLGPTN.

clr_flg Clear Eventflag [S]

[C Language API]

ER ercd = clr_flg (ID flgid, FLGPTN clrptn) ;

[Parameter]

ID	flgid	ID number of the eventflag to be cleared
FLGPTN	clrptn	Bit pattern to clear (bit-wise negated)

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (flgid is invalid or unusable)
E_NOEXS	Non-existent object (specified eventflag is not registered)
E_PAR	Parameter error (clrptn is invalid)

[Functional Description]

This service call clears the bits in the eventflag specified by flgid that correspond to 0 bit in clrptn. Specifically, the bit pattern of the eventflag is updated to the bit-wise AND of its bit pattern before the invocation of the service call with the value specified in clrptn.

[Supplemental Information]

No action is required when all of the bits of clrptn are 1.

[Differences from the µITRON3.0 Specification]

The data type of clrptn has been changed from UINT to FLGPTN.

wai_flg	Wait for Eventflag	[S]
pol_flg	Wait for Eventflag (Polling)	[S]
twai_flg	Wait for Eventflag (with Timeout)	[S]

[C Language API]

```
ER ercd = wai_flg ( ID flgid, FLGPTN waiptn, MODE wfmode,
                        FLGPTN *p_flgptn ) ;
ER ercd = pol_flg ( ID flgid, FLGPTN waiptn, MODE wfmode,
                        FLGPTN *p_flgptn ) ;
ER ercd = twai_flg ( ID flgid, FLGPTN waiptn, MODE wfmode,
                        FLGPTN *p_flgptn, TMO tmout ) ;
```

[Parameter]

ID	flgid	ID number of the eventflag to wait for
FLGPTN	waiptn	Wait bit pattern
MODE	wfmode	Wait mode
TMO	tmout	Specified timeout (only twai_flg)

[Return Parameter]

ER	ercd	E_OK for normal completion or error code
FLGPTN	flgptn	Bit pattern causing a task to be released from waiting

[Error Code]

E_ID	Invalid ID number (flgid is invalid or unusable)
E_NOEXS	Non-existent object (specified eventflag is not registered)
E_PAR	Parameter error (waiptn, wfmode, p_flgptn, or tmout is invalid)
E_ILUSE	Illegal service call use (there is already a task waiting for an eventflag with the TA_WSGL attribute)
E_RLWAI	Forced release from waiting (accept rel_wai while waiting; except pol_flg)
E_TMOUT	Polling failure or timeout (except wai_flg)
E_DLT	Waiting object deleted (eventflag is deleted while waiting; except pol_flg)

[Functional Description]

These service calls cause invoking task to wait until the eventflag specified by flgid satisfies the release condition. The release condition is determined by the bit pattern specified by waiptn and the wait mode specified by wfmode. Once the release condition is satisfied, the bit pattern causing the release is returned through flgptn. Specifically, the following actions are performed.

If the release condition is already satisfied when the service calls are invoked, the service calls complete without causing the invoking task to wait. The eventflag bit pattern is still returned to the invoking task through flgptn. In addition, when the eventflag's attribute has TA_CLR set, all the bits in the eventflag's bit pattern are cleared.

If the release condition is not satisfied, the invoking task is placed in the eventflag's wait queue and is moved to the waiting state for the eventflag.

When the eventflag's attribute has TA_WSGL (= 0x00) set and another task is already waiting in the eventflag's wait queue, an E_ILUSE error is returned. This applies even if the release condition is already satisfied.

wfmode can be specified as (TWF_ANDW || TWF_ORW). When wfmode has TWF_ANDW (= 0x00) set, the release condition requires all the bits in waiptn to be set. Conversely, when wfmode has TWF_ORW (= 0x01) set, the release condition only requires at least one bit in waiptn to be set.

If there are already tasks in the wait queue, the invoking task is placed in the wait queue as described below. When the eventflag's attribute has TA_TFIFO (= 0x00) set, the invoking task is placed in the tail of the wait queue. When the attribute has TA_TPRI (= 0x01) set, the invoking task is placed in the wait queue in the order of the task's priority. If the wait queue contains tasks with the same priority as the invoking task, the invoking task is placed after those tasks.

pol_flg is a polling service call with the same functionality as wai_flg. twai_flg has the same functionality as wai_sem with an additional timeout feature. tmout can be set to a positive number indicating a timeout duration or it can be set to TMO_POL (= 0) or TMO_FEVR (= –1).

If waitpn is 0, an E_PAR error is returned.

[Supplemental Information]

twai_flg acts the same as pol_flg if TMO_POL is specified in tmout as long as no context error occurs. Also, twai_flg acts the same as wai_flg if TMO_FEVR is specified in tmout.

[Differences from the µITRON3.0 Specification]

The order of parameters and the return parameter have been changed. The data type of waiptn and flgptn has been changed from UINT to FLGPTN, and the data type of wfmode has been changed from UINT to MODE.

The clear specification in the wait mode (TWF_CLR) has been removed. Instead, an eventflag attribute TA_CLR has been added. The value of TWF_ORW has been changed.

[Rationale]

The reason that an E_PAR error is returned when waiptn is 0 is because the release condition will never be satisfied.

ref_flg Reference Eventflag Status

[C Language API]

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

[Parameter]

ID	flgid	ID number of the eventflag to be referenced
T_RFLG *	pk_rflg	Pointer to the packet returning the eventflag state

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

pk_rflg includes (T_RFLG type)

ID	wtskid	ID number of the task at the head of the eventflag's wait queue
FLGPTN	flgptn	Eventflag's current bit pattern

(Other implementation specific information may be added.)

[Error Code]

E_ID	Invalid ID number (flgid is invalid or unusable)
E_NOEXS	Non-existent object (specified eventflag is not registered)
E_PAR	Parameter error (pk_rflg is invalid)

[Functional Description]

This service call references the state of the eventflag specified by parameter flgid. The state of the eventflag is returned through the packet pointed to by pk_rflg.

The ID number of the task at the head of the eventflag's wait queue is returned through wtskid. If no tasks are waiting for the eventflag, TSK_NONE (= 0) is returned instead.

The eventflag's current bit pattern is returned through flgptn.

[Differences from the µITRON3.0 Specification]

The extended information has been removed from the reference information. The ID number of the task at the head of the wait queue is now returned instead of a boolean value indicating whether a task is waiting or not. Based on this replacement, the name and data type of the return parameter has been changed.

The data type of flgptn has been changed from UINT to FLGPTN. The order of parameters and of return parameters has been changed.

4.4.3 Data Queues

A data queue is an object used for synchronization and communication by sending or receiving a one word message, called a data element. Data queue functions include the ability to create and delete a data queue, to send, force-send and receive a data element to/from a data queue, and to reference the state of a data queue. A data queue is an object identified by an ID number. The ID number of a data queue is called the data queue ID.

A data queue has an associated wait queue for sending a data element (send-wait queue) and an associated wait queue for receiving a data element (receive-wait queue). Also, a data queue has an associated data queue area used to store sent data elements. A task sending a data element (notifying the occurrence of an event) places the data element in the data queue. If there is no room in the data queue area, the task will be in the sending waiting state for a data queue until there is room for the data element in the data queue area. The task waiting to send the data element is placed in the data queue's send-wait queue. A task receiving a data element (waiting for an occurrence of an event) removes a data element from the data queue. If there is no data in the data queue, the task will be in the receiving waiting state until a data element is sent to the data queue. The task waiting to receive a data element from the data queue is placed in the data queue's receive-wait queue.

Synchronous message passing can be performed by setting the number of data elements that can be stored in the data queue area to 0. The sending task and the receiving task wait until the other calls the complimentary service call, at which time the data element is transferred.

The one word data element to be sent and received can be an integer or the address of a message located in a memory area shared by the sender and the receiver. A data element that is sent and received is copied from the sender to the receiver.

The following kernel configuration macro is defined for use with the data queue functions:

```
SIZE dtqsz = TSZ_DTQ ( UINT dtqcnt )
```

This macro returns the total required size of the data queue area in bytes necessary to store dtqcnt data elements.

The following date types packets are defined for creating and referencing data queues:

```
typedef struct t_cdtq {
        ATR       dtqatr ;     /* Data queue attribute */
        UINT      dtqcnt ;     /* Capacity of the data queue area (the
                                  number of data elements) */
        VP        dtq ;        /* Start address of the data queue area */
        /* Other implementation specific fields may be added. */
} T_CDTQ ;

typedef struct t_rdtq {
```

```
        ID        stskid ;     /* ID number of the task at the head of the
                                  data queue's send-wait queue */
        ID        rtskid ;     /* ID number of the task at the head of the
                                  data queue's receive-wait queue */
        UINT      sdtqcnt ;    /* The number of data elements in the data
                                  queue */
        /* Other implementation specific fields may be added. */
} T_RDTQ ;
```

The following represents the function codes for the data queue service calls:

TFN_CRE_DTQ	–0x31	Function code of cre_dtq
TFN_ACRE_DTQ	–0xc4	Function code of acre_dtq
TFN_DEL_DTQ	–0x32	Function code of del_dtq
TFN_SND_DTQ	–0x35	Function code of snd_dtq
TFN_PSND_DTQ	–0x36	Function code of psnd_dtq
TFN_IPSND_DTQ	–0x77	Function code of ipsnd_dtq
TFN_TSND_DTQ	–0x37	Function code of tsnd_dtq
TFN_FSND_DTQ	–0x38	Function code of fsnd_dtq
TFN_IFSND_DTQ	–0x78	Function code of ifsnd_dtq
TFN_RCV_DTQ	–0x39	Function code of rcv_dtq
TFN_PRCV_DTQ	–0x3a	Function code of prcv_dtq
TFN_TRCV_DTQ	–0x3b	Function code of trcv_dtq
TFN_REF_DTQ	–0x3c	Function code of ref_dtq

[Standard Profile]

The Standard Profile requires support for data queue functions except for dynamic creation and deletion of a data queue (cre_dtq, acre_dtq, del_dtq) and reference of a data queue state (ref_dtq).

The Standard Profile does not require TSZ_DTQ to be defined.

[Supplemental Information]

Figure 4-1 shows the behavior of a data queue when the number of data elements that can be stored in the data queue is set to 0. In this figure, task A and task B are assumed to be running asynchronously.

- If task A invokes snd_dtq first, task A is moved to the WAITING state until task B invokes rcv_dtq. During this time, task A is in the sending waiting state for a data queue.
- If, on the other hand, task B invokes rcv_dtq first, task B is moved to the WAITING state until task A invokes snd_dtq. During this time, task B is in the receiving waiting state for a data queue.
- When task A invokes snd_dtq and task B invokes rcv_dtq, the data transfer from task A to task B takes place. After this, both tasks are moved to the runnable state.

A data queue is assumed to be implemented as a ring buffer.

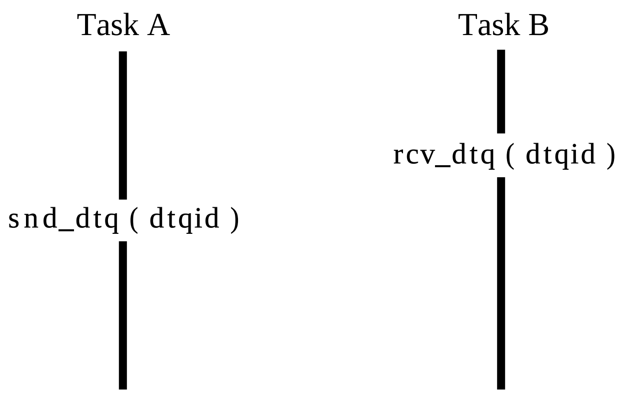


Figure 4-1. Synchronous Communication through a Data Queue

[Differences from the µITRON3.0 Specification]

This functionality has been newly added and has the same functionality as the mailbox of the µITRON3.0 Specification implemented with a ring buffer.

CRE_DTQ	Create Data Queue (Static API)	[S]
cre_dtq	Create Data Queue	
acre_dtq	Create Data Queue (ID Number Automatic Assignment)	

[Static API]

CRE_DTQ (ID dtqid, { ATR dtqatr, UINT dtqcnt, VP dtq }) ;

[C Language API]

ER ercd = cre_dtq (ID dtqid, T_CDTQ *pk_cdtq) ;
ER_ID dtqid = acre_dtq (T_CDTQ *pk_cdtq) ;

[Parameter]

ID	dtqid	ID number of the data queue to be created (except acre_dtq)
T_CDTQ *	pk_cdtq	Pointer to the packet containing the data queue creation information (in CRE_DTQ, packet contents must be directly specified.)

pk_cdtq includes (T_CDTQ type)

ATR	dtqatr	Data queue attribute
UINT	dtqcnt	Capacity of the data queue area (the number of data elements)
VP	dtq	Start address of the data queue area

(Other implementation specific information may be added.)

[Return Parameter]

cre_dtq:

ER	ercd	E_OK for normal completion or error code

acre_dtq:

ER_ID	dtqid	ID number (positive value) of the created data queue or error code

[Error Code]

E_ID	Invalid ID number (dtqid is invalid or unusable; only cre_dtq)
E_NOID	No ID number available (there is no data queue ID assignable; only acre_dtq)
E_NOMEM	Insufficient memory (data queue area cannot be allocated)
E_RSATR	Reserved attribute (dtqatr is invalid or unusable)
E_PAR	Parameter error (pk_cdtq, dtqcnt, or dtq is invalid)
E_OBJ	Object state error (specified data queue is already registered; only cre_dtq)

[Functional Description]

These service calls create a data queue with an ID number specified by dtqid based on the information contained in the packet pointed to by pk_cdtq. dtqatr is the attribute of the data queue. dtqcnt is the capacity of the data queue area: the maximum number of data elements that may be stored in the data queue area. dtq is the start address of the data queue area.

In CRE_DTQ, dtqid is an integer parameter with automatic assignment. dtqatr is a preprocessor constant expression parameter.

acre_dtq assigns a data queue ID from the pool of unassigned data queue IDs and returns the assigned data queue ID.

dtqatr can be specified as (TA_TFIFO || TA_TPRI). If TA_TFIFO (= 0x00) is specified, the data queue's send-wait queue will be in FIFO order. If TA_TPRI (= 0x01) is specified, the data queue's send-wait queue will be in task priority order.

The necessary area to hold up to dtqcnt data elements starts from dtq and is used as the data queue area. An application program can calculate the size of the data queue area necessary to hold dtqcnt number of data elements by using the TSZ_DTQ macro. If dtq is NULL (= 0), the kernel allocates the necessary memory area. dtqcnt may be specified as 0.

[Standard Profile]

The Standard Profile does not require support for when other values than NULL are specified in dtq.

[Supplemental Information]

The data queue's receive-wait queue always utilizes FIFO ordering. Also, a data element sent to a data queue does not have a priority. The data elements in a data queue is always in FIFO order. However, when snd_dtq and fsnd_dtq are used at the same time, there are cases where the data element sent by fsnd_dtq would be ahead of the data element earlier sent by snd_dtq.

del_dtq Delete Data Queue

[C Language API]

ER ercd = del_dtq (ID dtqid) ;

[Parameter]

ID	dtqid	ID number of the data queue to be deleted

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (dtqid is invalid or unusable)
E_NOEXS	Non-existent object (specified data queue is not registered)

[Functional Description]

This service call deletes the data queue specified by dtqid. If the data queue area was allocated by the kernel, the area is released.

[Supplemental Information]

The data elements in the data queue will be discarded. See Section 3.8 for information regarding handling tasks that are waiting in the data queue's send-wait queue and receive-wait queue when the data queue is deleted.

snd_dtq	Send to Data Queue	[S]
psnd_dtq	Send to Data Queue (Polling)	[S]
ipsnd_dtq		[S]
tsnd_dtq	Send to Data Queue (with Timeout)	[S]

[C Language API]

```
ER ercd = snd_dtq ( ID dtqid, VP_INT data ) ;
ER ercd = psnd_dtq ( ID dtqid, VP_INT data ) ;
ER ercd = ipsnd_dtq ( ID dtqid, VP_INT data ) ;
ER ercd = tsnd_dtq ( ID dtqid, VP_INT data, TMO tmout ) ;
```

[Parameter]

ID	dtqid	ID number of the data queue to which the data element is sent
VP_INT	data	Data element to be sent
TMO	tmout	Specified timeout (only tsnd_dtq)

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (dtqid is invalid or unusable)
E_NOEXS	Non-existent object (specified data queue is not registered)
E_PAR	Parameter error (tmout is invalid; only tsnd_dtq)
E_RLWAI	Forced release from waiting (accept rel_wai while waiting; only snd_dtq and tsnd_dtq)
E_TMOUT	Polling failure or timeout (except snd_dtq)
E_DLT	Waiting object deleted (data queue is deleted while waiting; only snd_dtq and tsnd_dtq)

[Functional Description]

These service calls send the data element specified by data to the data queue specified by dtqid. Specifically, the following actions are performed.

If there are already tasks in the data queue's receive-wait queue, these service calls send the data element to the task at the head of the receive-wait queue and release the task from waiting. The released task receives E_OK from the service call that caused it to wait in the receive-wait queue. It also receives the data element from the data queue through data.

If no tasks are waiting in the data queue's receive-wait queue, these service calls place the data element to be sent at the tail of the data queue. If there is no room in the data queue area, the invoking task is placed in the send-wait queue and is moved to the sending waiting state for the data queue.

If there are already tasks in the send-wait queue, the invoking task is placed in the send-wait queue as described below. When the data queue's attribute has TA_FIFO (= 0x00) set, the invoking task is placed at the tail of the send-wait queue. When the data queue's attribute has TA_TPRI (= 0x01) set, the invoking task is placed in the send-wait queue in the order of the task's priority. If the send-wait queue contains tasks with the same priority as the invoking task, the invoking task is placed after those tasks.

psnd_dtq and ipsnd_dtq are polling service calls with the same functionality as snd_dtq. tsnd_dtq has the same functionality as snd_dtq with an additional time-out feature. tmout can be set to a positive number indicating a timeout duration or it can be set to TMO_POL (= 0) or TMO_FEVR (= –1).

psnd_dtq and ipsnd_dtq return an E_TMOUT error if no tasks are waiting in the receive-wait queue and if there is no room for the data element in the data queue area. If the service call is invoked from non-task contexts and has its execution delayed, an E_TMOUT error may not be returned.

[Supplemental Information]

tsnd_dtq acts the same as psnd_dtq if TMO_POL is specified in tmout as long as no context error occurs. Also, tsnd_dtq acts the same as snd_dtq if TMO_FEVR is specified in tmout.

fsnd_dtq Forced Send to Data Queue [S]
ifsnd_dtq [S]

[C Language API]

```
ER ercd = fsnd_dtq ( ID dtqid, VP_INT data ) ;
ER ercd = ifsnd_dtq ( ID dtqid, VP_INT data ) ;
```

[Parameter]

ID	dtqid	ID number of the data queue to which the data element is sent
VP_INT	data	Data element to be sent to the data queue

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (dtqid is invalid or unusable)
E_NOEXS	Non-existent object (specified data queue is not registered)
E_ILUSE	Illegal service call use (the capacity of the data queue area is 0)

[Functional Description]

These service calls forcibly send the data element specified by data to the data queue specified by dtqid. Specifically, the following actions are performed.

If there are already tasks in the data queue's receive-wait queue, these service calls send the data element to the task at the head of the receive-wait queue and release the task from waiting. The released task receives E_OK from the service call that caused it to wait in the receive-wait queue. It also receives the data element from the data queue through data.

If no tasks are waiting in the data queue's receive-wait queue, these service calls place the data element to be sent at the tail of the data queue. If there is no room in the data queue area, these service calls reserve a space for the new data element by deleting the first data element in the data queue. The new data element is still placed at the tail of the data queue.

These service calls cannot forcibly send a data element when the capacity of the data queue area is 0. If the capacity of the data queue area is 0, an E_ILUSE error is returned.

[Supplemental Information]

These service calls force the data to be sent even if there are already tasks waiting to send data in the send-wait queue.

If the capacity of the data queue area is 0, an E_ILUSE error is returned even if there is a task waiting in the receive-wait queue.

rcv_dtq	Receive from Data Queue	[S]
prcv_dtq	Receive from Data Queue (Polling)	[S]
trcv_dtq	Receive from Data Queue (with Timeout)	[S]

[C Language API]

```
ER ercd = rcv_dtq ( ID dtqid, VP_INT *p_data ) ;
ER ercd = prcv_dtq ( ID dtqid, VP_INT *p_data ) ;
ER ercd = trcv_dtq ( ID dtqid, VP_INT *p_data, TMO tmout ) ;
```

[Parameter]

ID	dtqid	ID number of the data queue from which a data element is received
TMO	tmout	Specified timeout (only trcv_dtq)

[Return Parameter]

ER	ercd	E_OK for normal completion or error code
VP_INT	data	Data element received from the data queue

[Error Code]

E_ID	Invalid ID number (dtqid is invalid or unusable)
E_NOEXS	Non-existent object (specified data queue is not registered)
E_PAR	Parameter error (p_data or tmout is invalid)
E_RLWAI	Forced release from waiting (accept rel_wai while waiting; except prcv_dtq)
E_TMOUT	Polling failure or timeout (except rcv_dtq)
E_DLT	Waiting object deleted (data queue is deleted while waiting; except prcv_dtq)

[Functional Description]

These service calls receive a data element from the data queue specified by dtqid and returns the data element through data. Specifically, the following actions are performed.

If the data queue already has data elements, these service calls remove the first data element from the data queue and return it through data. If there is a task in the data queue's send-wait queue, these service calls place the data element from the first task in the send-wait queue at the tail of the data queue and release the task from waiting. The released task receives E_OK from the service call that caused it to wait in the send-wait queue.

If there are no data elements in the data queue and if there are tasks in the data queue's send-wait queue (this occurs when the capacity of the data queue area is 0), the data element from the task at the head of the send-wait queue is returned through data, and the task is released from waiting. The released task receives E_OK from the service call that caused it to wait in the send-wait queue.

If there are no data elements in the data queue and if there are no tasks in the send-wait queue, the invoking task is placed in the receive-wait queue and moved to the receiving waiting state for the data queue. If there are already tasks in the receive-wait queue, the invoking task is placed at the tail of the receive-wait queue.

prcv_dtq is a polling service call with the same functionality as rcv_dtq. trcv_dtq has the same functionality as rcv_dtq with an additional timeout feature. tmout can be set to a positive number indicating a timeout duration or it can be set to TMO_POL (= 0) or TMO_FEVR (= –1).

[Supplemental Information]

trcv_dtq acts the same as prcv_dtq if TMO_POL is specified in tmout as long as no context error occurs. Also, trcv_dtq acts the same as rcv_dtq if TMO_FEVR is specified in tmout.

ref_dtq Reference Data Queue State

[C Language API]

```
ER ercd = ref_dtq ( ID dtqid, T_RDTQ *pk_rdtq ) ;
```

[Parameter]

ID	dtqid	ID number of the data queue to be referenced
T_RDTQ *	pk_rdtq	Pointer to the packet returning the data queue state

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

pk_rdtq includes (T_RDTQ type)

ID	stskid	ID number of the task at the head of the send-wait queue
ID	rtskid	ID number of the task at the head of the receive-wait queue
UINT	sdtqcnt	The number of data elements in the data queue

(Other implementation specific information may be added.)

[Error Code]

E_ID	Invalid ID number (dtqid is invalid or unusable)
E_NOEXS	Non-existent object (specified data queue is not registered)
E_PAR	Parameter error (pk_rdtq is invalid)

[Functional Description]

This service call references the state of the data queue specified by dtqid. The state of the data queue is returned through the packet pointed to by pk_rdtq.

The ID number of the task at the head of the data queue's send-wait queue is returned through stskid. If no tasks are waiting to send a data element, TSK_NONE (= 0) is returned instead.

The ID number of the task at the head of the data queue's receive-wait queue is returned through rtskid. If no tasks are waiting to receive a data element, TSK_NONE (= 0) is returned instead.

The number of data elements currently in the data queue is returned through sdtqcnt.

[Supplemental Information]

A data queue cannot have rtskid ≠ TSK_NONE and sdtqcnt ≠ 0 at the same time. When stskid ≠ TSK_NONE, sdtqcnt is equal to the capacity of the data queue area.

4.4.4 Mailboxes

A mailbox is an object used for synchronization and communication by sending or receiving a message placed in a shared memory. Mailbox functions include the ability to create and delete a mailbox, to send and receive a message to/from a mailbox, and to reference the state of a mailbox. A mailbox is an object identified by an ID number. The ID number of a mailbox is called the mailbox ID.

A mailbox has an associated message queue used to store sent messages and an associated wait queue for receiving messages. A task sending a message (notifying the occurrence of an event) places the message to be sent in the message queue. A task receiving a message from the mailbox (waiting for an occurrence of an event) removes the first message from the message queue. If there is no message in the message queue, the task will be in the receiving waiting state until a message is sent to the mailbox. The task waiting to receive a message from the mailbox is placed in the mailbox's wait queue.

With mailbox functions, only the start address of the message placed in a shared memory is actually passed between tasks. The message itself is not copied.

The kernel maintains the messages in the message queue using a linked list. The application program must reserve an area to be used by the kernel for the linked list at the head of each sent message. This reserved area is called the message header. A message packet is the area consisting of a message header followed by an area that is used by the application to store a message body. The start address of the message packet is passed as a parameter to the service calls that send a message, and it is returned as a return parameter from the service calls that receive a message. An area for the message priority is included in the message header when the message queue is ordered by message priorities.

The kernel modifies the contents of a message header, except the area for the message priority, while the message is in a message queue (and when the message is to be placed in a message queue). On the other hand, the application program must not modify the contents of a message header, including the message priority, while the message is in a message queue. If the application modifies the contents of a message header, the resulting behavior is undefined. In addition to the case where the application program directly modifies the contents of a message header, this rule also applies to the case where the application program passes the address of the message header to the kernel and makes the kernel modify its contents. Therefore, the behavior when a message already in a message queue is resent to a mailbox is undefined.

The following data types are used for message headers:

T_MSG	Message header for a mailbox
T_MSG_PRI	Message header with a message priority for a mailbox

The definition and size of the T_MSG type are implementation-defined. The

T_MSG_PRI type is defined using T_MSG type as follows:

```
typedef struct t_msg_pri {
        T_MSG       msgque ;    /* Message header */
        PRI         msgpri ;    /* Message priority */
} T_MSG_PRI ;
```

The following kernel configuration macro is defined for use with mailbox functions:

SIZE mprihdsz = TSZ_MPRIHD (PRI maxmpri)

This macro returns the total required size in bytes of the area for message queue headers for each message priority, when the maximum message priority is maxmpri.

The following data types packets are defined for creating and referencing mailboxes:

```
typedef struct t_cmbx {
        ATR         mbxatr ;    /* Mailbox attribute */
        PRI         maxmpri ;   /* Maximum message priority */
        VP          mprihd ;    /* Start address of the area for message
                                   queue headers for each message
                                   priority */
        /* Other implementation specific fields may be added. */
} T_CMBX ;

typedef struct t_rmbx {
        ID          wtskid ;    /* ID number of the task at the head of
                                   mailbox's wait queue */
        T_MSG *     pk_msg ;    /* Start address of the message packet at
                                   the head of the message queue */
        /* Other implementation specific fields may be added. */
} T_RMBX ;
```

The following represents the function codes for the mailbox service calls:

TFN_CRE_MBX	–0x3d	Function code of cre_mbx
TFN_ACRE_MBX	–0xc5	Function code of acre_mbx
TFN_DEL_MBX	–0x3e	Function code of del_mbx
TFN_SND_MBX	–0x3f	Function code of snd_mbx
TFN_RCV_MBX	–0x41	Function code of rcv_mbx
TFN_PRCV_MBX	–0x42	Function code of prcv_mbx
TFN_TRCV_MBX	–0x43	Function code of trcv_mbx
TFN_REF_MBX	–0x44	Function code of ref_mbx

[Standard Profile]

The Standard Profile requires support for mailbox functions except for dynamic creation and deletion of a mailbox (cre_mbx, acre_mbx, del_mbx) and reference of a mailbox state (ref_mbx).

The Standard Profile does not require TSZ_MPRIHD to be defined.

[Supplemental Information]

In the mailbox functions, the number of messages that can be stored in a message queue has no upper limit because the application program has the responsibility to reserve the area for message headers. Service calls for sending a message will not move the invoking task to the WAITING state.

To make an application program portable to different kernels with different definitions and sizes for message headers, the message packet should be defined as a C language structure, and a field of T_MSG type or T_MSG_PRI type should be allocated at the top of the message packet. In addition the message priority should be assigned to the msgpri field in T_MSG_PRI type. sizeof (T_MSG) or sizeof (T_MSG_PRI) can be used to determine the size of the message header.

The area for a message packet may be dynamically allocated from a fixed-sized or variable-sized memory pool. It is also possible to allocate the area statically. A common practice is that the sending task allocates a memory block from a memory pool and sends the block as a message packet to a mailbox, while the receiving task releases the memory block which is received as a message packet from the mailbox to the memory pool.

[Differences from the μITRON3.0 Specification]

Implementations of mailboxes are now limited to linked lists.

CRE_MBX	Create Mailbox (Static API)	[S]
cre_mbx	Create Mailbox	
acre_mbx	Create Mailbox (ID Number Automatic Assignment)	

[Static API]

CRE_MBX (ID mbxid, { ATR mbxatr, PRI maxmpri,
VP mprihd }) ;

[C Language API]

ER ercd = cre_mbx (ID mbxid, T_CMBX *pk_cmbx) ;
ER_ID mbxid = acre_mbx (T_CMBX *pk_cmbx) ;

[Parameter]

ID	mbxid	ID number of the mailbox to be created (except: acre_mbx)
T_CMBX *	pk_cmbx	Pointer to the packet containing the mailbox creation information (in CRE_MBX, packet contents must be directly specified.)

pk_cmbx includes (T_CMBX type)

ATR	mbxatr	Mailbox attribute
PRI	maxmpri	Maximum message priority
VP	mprihd	Start address of the area for message queue headers for each message priority

(Other implementation specific information may be added.)

[Return Parameter]

cre_mbx:

ER	ercd	E_OK for normal completion or error code

acre_mbx:

ER_ID	mbxid	ID number (positive value) of created mailbox or error code

[Error Code]

E_ID	Invalid ID number (mbxid is invalid or unusable; only cre_mbx)
E_NOID	No ID number available (there is no mailbox ID assignable; only acre_mbx)
E_NOMEM	Insufficient memory (message queue header area cannot be allocated)
E_RSATR	Reserved attribute (mbxatr is invalid or unusable)
E_PAR	Parameter error (pk_cmbx, maxmpri, or mprihd is invalid)

E_OBJ Object state error (specified mailbox is already registered; only cre_mbx)

[Functional Description]

These service calls create a mailbox with ID number specified by mbxid based on the information contained in the packet pointed to by pk_cmbx. mbxatr is the attribute of the mailbox. maxmpri is the maximum message priority of messages sent to the mailbox. mprihd is the start address of the area for message queue headers for each message priority. maxmpri and mprihd are valid only when TA_MPRI (= 0x02) is specified in mbxatr.

In CRE_MBX, mbxid is an integer parameter with automatic assignment. mbxatr and maxmpri are preprocessor constant expression parameters.

acre_mbx assigns a mailbox ID from the pool of unassigned mailbox IDs and and returns the asigned mailbox ID.

mbxatr can be specified as ((TA_TFIFO || TA_TPRI) | (TA_MFIFO || TA_MPRI)). If TA_TFIFO (= 0x00) is specified, the mailbox's wait queue will be in FIFO order. If TA_TPRI (= 0x01) is specified, the mailbox's wait queue will be in task priority order. Similarly, if TA_MFIFO (= 0x00) is specified, the mailbox's message queue will be in FIFO order, and if TA_MPRI (= 0x02) is specified, the message queue will be in message priority order.

If TA_MPRI is specified in mbxatr, the necessary area to hold the message queue headers for each of the message priorities up to maxmpri starts from mprihd. An application program can calculate the size of the necessary message queue header area when the maximum message priority is maxmpri by using TSZ_MPRIHD macro. If mprihd is NULL (= 0), the kernel allocates the necessary memory area. maxmpri cannot be specified as 0. If specified, an E_PAR error is returned.

[Standard Profile]

The Standard Profile does not require support for when other values than NULL is specified in mprihd.

[Supplemental Information]

The following must be considered when a message queue is prepared for each message priority level using the message queue header area.

Preparing a message queue for each message priority level is effective when the number of the message priority levels is small. When the number of allowed message priority levels is large, this method requires a large memory area and thus is not practical. Therefore, in order to handle the case where the message priority levels is large, the structure of the message queue should be varied depending on the number of message priority levels. For example, when the maximum priority level is below a certain threshold value, a message queue is prepared for each message priority level. When the maximum priority level falls above this threshold, all messages are managed in a single queue. In this case, TSZ_MPRIHD will return the same value for all values of maxmpri that are above the threshold value. maxmpri parameter to CRE_MBX is defined to be a preprocessor constant expression parameter in order for the kernel configurator to create conditional directives involving maxmpri in the C language source code and to modify the data structure in the kernel when maxmpri is above the threshold value.

It is also possible to manage all messages in a single queue without using separate message queues for each message priority. In this kind of implementations, TSZ_MPRIHD should be defined so that it returns a constant value, regardless of maxmpri.

[Differences from the μITRON3.0 Specification]

The maximum message priority (maxmpri) and the start address of the area for message queue headers for each message priority (mprihd) have been added to the mailbox creation information. The extended information and the ring buffer size (an implementation-dependent information) have been removed.

acre_mbx has been newly aded.

del_mbx Delete Mailbox

[C Language API]

ER ercd = del_mbx (ID mbxid) ;

[Parameter]

ID	mbxid	ID number of the mailbox to be deleted

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (mbxid is invalid or unusable)
E_NOEXS	Non-existent object (specified mailbox is not registered)

[Functional Description]

This service call deletes the mailbox specified by mbxid. If the area for message queue headers for each message priority was allocated by the kernel, it is released.

[Supplemental Information]

The messages in the message queue will be discarded. See Section 3.8 for information regarding handling tasks that are waiting to receive a message in a mailbox's wait queue when the mailbox is deleted.

snd_mbx Send to Mailbox [S]

[C Language API]

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

[Parameter]

ID	mbxid	ID number of the mailbox to which the message is sent
T_MSG *	pk_msg	Start address of the message packet to be sent to the mailbox

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (mbxid is invalid or unusable)
E_NOEXS	Non-existent object (specified mailbox is not registered)
E_PAR	Parameter error (pk_msg is invalid, the message priority in the message packet (msgpri) is invalid)

[Functional Description]

This service call sends the message whose start address is specified by pk_msg to the mailbox specified by mbxid. Specifically, the following actions are performed.

If there are already tasks in the mailbox's wait queue, this service call sends the start address of the message packet to the task at the head of the wait queue and releases the task from waiting. The released task receives E_OK from the service call that caused it to wait in the wait queue. It also receives the start address of the message packet from the mailbox through pk_msg.

If no tasks are waiting in the mailbox's wait queue, this service call places the message packet to the message queue. When the mailbox's attribute has TA_MFIFIO (= 0x00) set, the message packet is placed at the tail of the message queue. When the mailbox's attribute has TA_MPRI (= 0x02) set, the message packet is placed in the message queue in the order of its message priority. If the message queue contains messages with the same priority as the newly sent message, the message is placed after those messages.

When the mailbox's attribute has TA_MPRI (= 0x02) set, the message header of T_MSG_PRI type is assumed to be at the head of the message packet pointed to by pk_msg. The message's priority is obtained from the msgpri field in the message header.

[Differences from the µITRON3.0 Specification]

The name of the service call has been changed from snd_msg into snd_mbx.

rcv_mbx	Receive from Mailbox	[S]
prcv_mbx	Receive from Mailbox (Polling)	[S]
trcv_mbx	Receive from Mailbox (with Timeout)	[S]

[C Language API]

```
ER ercd = rcv_mbx ( ID mbxid, T_MSG **ppk_msg ) ;
ER ercd = prcv_mbx ( ID mbxid, T_MSG **ppk_msg ) ;
ER ercd = trcv_mbx ( ID mbxid, T_MSG **ppk_msg,
                                  TMO tmout ) ;
```

[Parameter]

ID	mbxid	ID number of the mailbox from which a message is received
TMO	tmout	Specified timeout (only trcv_mbx)

[Return Parameter]

ER	ercd	E_OK for normal completion or error code
T_MSG *	pk_msg	Start address of the message packet received from the mailbox

[Error Code]

E_ID	Invalid ID number (mbxid is invalid or unusable)
E_NOEXS	Non-existent object (specified mailbox is not registered)
E_PAR	Parameter error (ppk_msg or tmout is invalid)
E_RLWAI	Forced release from waiting (accept rel_wai while waiting; except prcv_mbx)
E_TMOUT	Polling failure or timeout (except rcv_mbx)
E_DLT	Waiting object deleted (mailbox is deleted while waiting; except prcv_mbx)

[Functional Description]

These service calls receive a message from the mailbox specified by mbxid and return its start address through pk_msg. Specifically, the following actions are performed.

If the mailbox's message queue already has messages, these service calls remove the first message packet from the message queue and return its start address through pk_msg.

If there are no messages in the message queue, the invoking task is placed in the wait queue and moved to the receiving waiting state for the mailbox.

If there are already tasks in the wait queue, the invoking task is placed in the wait queue as described below. When the mailbox's attribute has TA_TFIFO (= 0x00) set, the invoking task is placed at the tail of the wait queue. When the mailbox's attribute has TA_TPRI (= 0x01) set, the invoking task is placed in the wait queue in the order of the task's priority. If the wait queue contains tasks with the same priority as the invoking task, the invoking task is placed after those tasks.

prcv_mbx is a polling service call with the same functionality as rcv_mbx. trcv_mbx has the same functionality as rcv_mbx with an additional timeout feature. tmout can be set to a positive number indicating a timeout duration or it can be set to TMO_POL (= 0) or TMO_FEVR (= –1).

[Supplemental Information]

trcv_mbx acts the same as prcv_mbx if TMO_POL is specified in tmout as long as no context error occurs. Also, trcv_mbx acts the same as rcv_mbx if TMO_FEVR is specified in tmout.

[Differences from the µITRON3.0 Specification]

The names of the service calls have been changed from rcv_msg, prcv_msg, trcv_msg to rcv_mbx, prcv_mbx, trcv_mbx, respectively. The order of parameters and of return parameters has been changed.

ref_mbx Reference Mailbox State

[C Language API]

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

[Parameter]

ID	mbxid	ID number of the mailbox to be referenced
T_RMBX *	pk_rmbx	Pointer to the packet returning the mailbox state

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

pk_rmbx includes (T_RMBX type)

ID	wtskid	ID number of the task at the head of the mailbox's wait queue
T_MSG *	pk_msg	Start address of the message packet at the head of message queue

(Other implementation specific information may be added.)

[Error Code]

E_ID	Invalid ID number (mbxid is invalid or unusable)
E_NOEXS	Non-existent object (specified mailbox is not registered)
E_PAR	Parameter error (pk_rmbx is invalid)

[Functional Description]

This service call references the state of the mailbox specified by mbxid. The state of the mailbox is returned through the packet pointed to by pk_rmbx.

The ID number of the task at the head of the mailbox's wait queue is returned through wtskid. If no tasks are waiting to receive a message, TSK_NONE (= 0) is returned instead.

The start address of the message packet at the head of the mailbox's message queue is returned through pk_msg. If there is no message in the message queue, NULL (= 0) is returned instead.

[Supplemental Information]

A mailbox cannot have wtskid ≠ TSK_NONE and pk_msg ≠ NULL at the same time.

[Differences from the µITRON3.0 Specification]

The extended information has been removed from the reference information. The ID number of the task at the head of the wait queue is now returned instead of a boolean value indicating whether a task is waiting or not. Based on this replacement, the name and data type of the return parameter has been changed.

The order of parameters and of return parameters has been changed.

4.5 Extended Synchronization and Communication Functions

Extend synchronization and communication functions provide advanced synchronization and communication between tasks through objects that are independent of the tasks. The objects are mutexes, message buffers, and rendezvous ports.

[Standard Profile]

The Standard Profile does not require support for extended synchronization and communication functions.

[Differences from the μITRON3.0 Specification]

Mutex is a newly added feature.

4.5.1 Mutexes

A mutex is an object used for mutual exclusion of a shared resource among tasks. Mutex supports the priority inheritance protocol and the priority ceiling protocol to avoid unbounded priority inversions among tasks competing for a shared resource. Mutex functions includes the ability to create and delete a mutex, to lock and unlock a mutex, and to reference the state of a mutex. A mutex is an object identified by an ID number. The ID number of a mutex is called the mutex ID.

A mutex has a locked and unlocked state. It also has a wait queue for tasks waiting to lock the mutex. The kernel manages the task that locks each mutex and also the set of mutexes a task locks. A task will try to lock a mutex before using a shared resource. In case a mutex is already locked by another task, the task will be placed in the WAITING state until the mutex is released. A task unlocks the mutex after using the shared resource.

A mutex uses the priority inheritance protocol when its attribute has TA_INHERIT (= 0x02) set, and it uses the priority ceiling protocol when its attribute has TA_CEILING (= 0x03) set. During mutex creation, if the TA_CEILING attribute is specified, the ceiling priority parameter should be set to the maximum priority of the tasks that may lock the mutex. When a task tries to lock a mutex with the TA_CEILING attribute and it has a higher base priority than the ceiling priority of the mutex, an E_ILUSE error is returned. If chg_pri is invoked to set the base priority of a task that has locked a mutex with the TA_CEILING attribute to a higher value than the mutex's ceiling priority, chg_pri will return an E_ILUSE error.

When using these protocols, mutex operations change the current priority of tasks in order to prevent unbounded priority inversion. The priority inheritance protocol and the priority ceiling protocol require that the current priority of a task should always be equal to the highest of the three priorities below:

- The base priority of the task
- The highest current priority among tasks waiting to lock one of the mutexes with the TA_INHERIT attribute that are locked by the task
- The highest ceiling priority among mutexes with the TA_CEILING attribute that are locked by the task

This rule is called the strict priority control rule.

If the current priority of a task waiting for a mutex with the TA_INHERIT attribute is changed by mutex operations or is changed by having its base priority changed by chg_pri, the task that has the mutex locked may have to have its current priority changed. Such a change of priority is called transitive priority inheritance. Moreover, if the latter task is waiting for a second mutex with the TA_INHERIT attribute, transitive priority inheritance needs to be applied to the task that has the second mutex locked.

In addition to the strict priority control rule, the μITRON4.0 Specification defines another priority control rule, called the simplified priority control rule, which limits the conditions under which the current priority is changed. The priority control rule used is implementation-defined. Under the simplified priority control rule, when the current priority of a task should be raised, it must be raised. However, when the current priority of a task should be lowered, it must be lowered only when the task no longer locks any mutexes. In the case where the current priority of the task is lowered, it is changed back to its base priority. More specifically, the current priority of a task is changed under the following conditions:

- When a higher-priority task begins to wait for a mutex with the TA_INHERIT attribute that is locked by the task.
- When the current priority of a task waiting for a mutex with the TA_INHERIT attribute that is locked by the task is changed to a higher priority than the task.
- When the task locks a mutex with the TA_CEILING attribute and with a higher ceiling priority than the task's current priority.
- When the task releases the last mutex that it locked.

The following actions are taken when the current priority of a task has been changed by mutex operations. When a task whose priority has been changed is in the runnable state, the precedence of the task is changed according to its new priority. The resulting precedence of the task among the tasks with the same priority is implementation-dependent. When a task whose priority has been changed is in a priority-ordered wait queue, the task's position in the wait queue is changed according to the new priority. The resulting position of the task among the tasks of the same priority is implementation-dependent.

If a task terminates while it still has mutexes locked, the kernel unlocks all the mutexes that it locked. The order of unlocking the mutexes is implementation-dependent. For more details about unlocking a mutex, see the functional description of unl_mtx.

The following data type packets are defined for creating and referencing mutexes:

```
typedef struct t_cmtx {
        ATR         mtxatr ;     /* Mutex attribute */
        PRI         ceilpri ;    /* Mutex ceiling priority */
        /* Other implementation specific fields may be added. */
} T_CMTX ;

typedef struct t_rmtx {
        ID          htskid ;     /* ID number of the task that locks the
                                    mutex */
        ID          wtskid ;     /* ID number of the task at the head of the
                                    mutex's wait queue */
        /* Other implementation specific fields may be added. */
} T_RMTX ;
```

The following represents the function codes for the mutex service calls:

TFN_CRE_MTX	–0x81	Function code of cre_mtx
TFN_ACRE_MTX	–0xc6	Function code of acre_mtx
TFN_DEL_MTX	–0x82	Function code of del_mtx
TFN_LOC_MTX	–0x85	Function code of loc_mtx
TFN_PLOC_MTX	–0x86	Function code of ploc_mtx
TFN_TLOC_MTX	–0x87	Function code of tloc_mtx
TFN_UNL_MTX	–0x83	Function code of unl_mtx
TFN_REF_MTX	–0x88	Function code of ref_mtx

[Supplemental Information]

A mutex with the attribute TA_TFIFO or TA_TPRI has a similar functionality as a semaphore whose maximum count is 1: a binary semaphore. The differences are that a mutex can only be unlocked by the task that locked it and that a mutex is unlocked by the kernel when the locking task terminates.

The definition of the priority ceiling protocol described here is different from the priority ceiling protocol proposed in literature. More strictly, this protocol is sometimes referred to as the highest locker protocol.

When mutex operations change the current priority of a task, and when the order of the task within a wait queue is changed, the kernel may need to release the task or other tasks in the wait queue from waiting. See the functional descriptions of snd_mbf and get_mpl for details.

[Differences from the μITRON3.0 Specification]

The mutex is newly added feature. Mutexes are introduced as objects independent from semaphores because supporting priority inheritance protocol for counting semaphores is difficult.

[Rationale]

When mutex operations change the current priority of a task, the precedence among the tasks with the same priority are made implementation-dependent for the following reasons. Some applications might require frequent changes of the current priority through the use of mutexes, resulting in frequent task switches, which in turn is not desirable. If precedence of the task among tasks of the same priority is determined to the lowest, unnecessary task switches may occur. Ideally, precedence (and not priority) should be inherited. However, such a specification would require a large overhead. For this reason, the precedence among tasks is left up to the implementation.

CRE_MTX	Create Mutex (Static API)
cre_mtx	Create Mutex
acre_mtx	Create Mutex (ID Number Automatic Assignment)

[Static API]

CRE_MTX (ID mtxid, { ATR mtxatr, PRI ceilpri }) ;

[C Language API]

ER ercd = cre_mtx (ID mtxid, T_CMTX *pk_cmtx) ;
ER_ID mtxid = acre_mtx (T_CMTX *pk_cmtx) ;

[Parameter]

ID	mtxid	ID number of the mutex to be created (except acre_mtx)
T_CMTX *	pk_cmtx	Pointer to the packet containing the mutex creation information (in CRE_MTX, the packet contents must be directly specified.)

pk_cmtx includes (T_CMTX type)

ATR	mtxatr	Mutex attribute
PRI	ceilpri	Mutex ceiling priority

(Other implementation specific information may be added.)

[Return Parameter]

cre_mtx:

ER	ercd	E_OK for normal completion or error code

acre_mtx:

ER_ID	mtxid	ID number (positive value) of the created mutex or error code

[Error Code]

E_ID	Invalid ID number (mtxid is invalid or unusable; only cre_mtx)
E_NOID	No ID number available (there is no mutex ID assignable; only acre_mtx)
E_RSATR	Reserved attribute (mtxatr is invalid or unusable)
E_PAR	Parameter error (pk_cmtx or ceilpri is invalid)
E_OBJ	Object state error (mutex is already registered; only cre_mtx)

[Functional Description]

These service calls create a mutex with an ID number specified by mtxid based on the information contained in the packet pointed to by pk_cmtx. mtxatr is the attribute of the mutex. ceilpri is the mutex ceiling priority. ceilpri is only valid when mtxatr has TA_CEILING (= 0x03) set.

In CRE_MTX, mtxid is an integer parameter with automatic assignment. mtxatr is a preprocessor constant expression parameter.

acre_mtx assigns a mutex ID from the pool of unassigned mutex IDs and returns the assigned mutex ID.

mtxatr can be specified as (TA_TFIFO || TA_TPRI || TA_INHERIT || TA_CEILING). If TA_FIFO (= 0x00) is specified, the mutex's wait queue will be in FIFO order. Otherwise, the mutex's wait queue will be in task priority order. If TA_INHERIT (= 0x02) is set, the current priority of a task is changed according to the priority inheritance protocol. If TA_CEILING (= 0x03) is set, the current priority of a task is changed according to the priority ceiling protocol.

del_mtx Delete Mutex

[C Language API]

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

[Parameter]

ID	mtxid	ID number of the mutex to be deleted

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (mtxid is invalid or unusable)
E_NOEXS	Non-existent object (specified mutex is not registered)

[Functional Description]

This service call deletes the mutex specified by mtxid.

[Supplemental Information]

If the specified mutex has been locked by a task, del_mtx forces the task to unlock the mutex it has locked. Therefore, if the mutex has either the TA_INHERIT or TA_CEILING attribute, the current priority of the task that has locked the mutex may need to be changed. When the simplified priority control rule is applied, the current priority of the locking task is changed only if after the deletion, no mutex remains locked by the task.

The task that locked the mutex is not notified about the deletion of the mutex. Rather, it will receive an error when it tries to unlock the mutex. If deleting a mutex will cause an undesirable result for the task that is locking the mutex, a task that tries to delete the mutex should first lock the mutex itself and then delete it.

See Section 3.8 for information regarding handling tasks that are waiting to lock a mutex when the mutex is deleted.

loc_mtx	Lock Mutex
ploc_mtx	Lock Mutex (Polling)
tloc_mtx	Lock Mutex (with Timeout)

[C Language API]

ER ercd = loc_mtx (ID mtxid) ;
ER ercd = ploc_mtx (ID mtxid) ;
ER ercd = tloc_mtx (ID mtxid, TMO tmout) ;

[Parameter]

ID	mtxid	ID number of the mutex to be locked
TMO	tmout	Specified timeout (only tloc_mtx)

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (mtxid is invalid or unusable)
E_NOEXS	Non-existent object (specified mutex is not registered)
E_PAR	Parameter error (tmout is invalid; only tloc_mtx)
E_ILUSE	Illegal service call use (multiple locking of a mutex, ceiling priority violation)
E_RLWAI	Forced release from waiting (accept rel_wai while waiting; except ploc_mtx)
E_TMOUT	Polling failure or timeout (except loc_mtx)
E_DLT	Waiting object deleted (mutex is deleted while waiting; except ploc_mtx)

[Functional Description]

These service calls lock the mutex specified by mtxid. Specifically, if the mutex is not locked, the service calls let the invoking task lock the mutex and return without moving the invoking task to the WAITING state. If the mutex is locked, the invoking task is placed in the mutex's wait queue and is moved to the waiting state for the mutex.

If there are already tasks in the wait queue, the invoking task is placed in the wait queue as described below. When the mutex's attribute has TA_TFIFO (= 0x00) set, the invoking task is placed at the tail of the wait queue. Otherwise, the invoking task is placed in the wait queue in the order of the task's priority. If the wait queue contains tasks with the same priority as the invoking task, the invoking task is placed after those tasks.

If the invoking task has already locked the mutex, these service calls return an E_ILUSE error. An E_ILUSE error will also be returned if the mutex has TA_CEILING attribute set and if the invoking task has a base priority higher than the ceiling priority of the mutex.

ploc_mtx is a polling service call with the same functionality as loc_mtx. tloc_mtx has the same functionality as loc_mtx with an additional timeout feature. tmout can be set to a positive number indicating a timeout duration or it can be set to TMO_POL (= 0) or TMO_FEVR (= –1).

[Supplemental Information]

When a task invokes these service calls on the mutex with TA_INHERIT attribute that is locked and is moved to the WAITING state, the current priority of the task that locks a mutex is changed to the current priority of the invoking task if the latter's current priority is lower than the current priority of the invoking task.

The current priority of a task that locks a mutex with TA_INHERIT attribute may need to be changed when a task that is waiting for the mutex is released from waiting due to a timeout or with rel_wai. The simplified priority control rule does not perform such a change.

When a task invokes these service calls on the mutex with TA_CEILING attribute and locks it successfully, the current priority of the task is changed to the ceiling priority of the mutex if the ceiling priority is higher than the task's current priority.

tloc_mtx acts the same as ploc_mtx if TMO_POL is specified in tmout as long as no context error occurs. Also, tloc_mtx acts the same as loc_mtx if TMO_FEVR is specified in tmout.

unl_mtx Unlock Mutex

[C Language API]

ER ercd = unl_mtx (ID mtxid) ;

[Parameter]

ID	mtxid	ID number of the mutex to be unlocked

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (mtxid is invalid or unusable)
E_NOEXS	Non-existent object (specified mutex is not registered)
E_ILUSE	Illegal service call use (the invoking task does not have the specified mutex locked)

[Functional Description]

This service call unlocks the mutex specified by mtxid. Specifically, if there are tasks waiting to lock the mutex, the service call release the task at the head of the mutex's wait queue from waiting and let the released task lock the mutex. The task receives E_OK from the service call that caused it to wait in the mutex's wait queue. If no task is waiting to lock the mutex, the service call moves the mutex to the unlocked state.

When the invoking task does not have the mutex locked, this service call returns an E_ILUSE error.

[Supplemental Information]

The current priority of the task invoking this service call may need to be changed when the specified mutex has the TA_INHERIT or TA_CEILING attribute set. If the simplified priority control rule is applied, the service call changes the current priority of the invoking task only when no mutex remains locked by the task.

ref_mtx Reference Mutex State

[C Language API]

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

[Parameter]

ID	mtxid	ID number of the mutex to be referenced
T_RMTX *	pk_rmtx	Pointer to the packet returning the mutex state

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

pk_rmtx includes (T_RMTX type)

ID	htskid	ID number of the task locking the mutex
ID	wtskid	ID number of the task at the head of the mutex's wait queue

(Other implementation specific information may be added.)

[Error Code]

E_ID	Invalid ID number (mtxid is invalid or unusable)
E_NOEXS	Non-existent object (specified mutex is not registered)
E_PAR	Parameter error (pk_rmtx is invalid)

[Functional Description]

This service call references the state of the mutex specified by mtxid. The state of the mutex is returned through the packet pointed to by pk_rmtx.

The ID number of the task that has the mutex locked is returned through htskid. If no task has the mutex locked, TSK_NONE (= 0) is returned instead.

The ID number of the task at the head of the mutex's wait queue is returned through wtskid. If no tasks are waiting to lock the mutex TSK_NONE (= 0) is returned instead.

[Supplemental Information]

A mutex cannot have htskid = TSK_NONE and wtskid ≠ TSK_NONE at the same time.

4.5.2 Message Buffers

A message buffer is an object used for synchronization and communication by sending and receiving a variable-sized message. Message buffer functions include the ability to create and delete a message buffer, to send and receive a message to/from a message buffer, and to reference the state of a message buffer. A message buffer is an object identified by an ID number. The ID number of a message buffer is called the message buffer ID.

A message buffer has an associated wait queue for sending a message (send-wait queue) and an associated wait queue for receiving a message (receive-wait queue). Also, a message buffer has an associated message buffer area to store the sent messages. A task sending a message (notifying the occurrence of an event) copies the message into the message buffer. If there is no room in the message buffer area, the task will be in the sending waiting state for a message buffer until there is room for the message in the message buffer area. The task waiting to send the message is placed in the message buffer's send-wait queue. A task receiving a message (waiting for an occurrence of an event) removes a message from the message buffer. If there is no message in the message buffer, the task will be in the receiving waiting state until a message is sent to the message buffer. The task waiting to receive a message from the message buffer is placed in the message buffer's receive-wait queue.

Synchronous message passing can be performed by setting the size of the message buffer area to 0. The sending task and the receiving task wait until the other calls the complimentary service call, at which time the message is transferred.

The following kernel configuration macro is defined for use with message buffer functions:

SIZE mbfsz = TSZ_MBF (UINT msgcnt, UINT msgsz)

> This macro returns the approximate required size of the message buffer area in bytes necessary to store msgcnt messages each consisting of msgsz bytes.

This macro is only an estimation for determining the size of a message buffer area. It cannot be used to determine the total required size of a message buffer area to store messages with different sizes.

The following data type packets are defined for creating and referencing message buffers:

```
typedef struct t_cmbf {
        ATR     mbfatr ;    /* Message buffer attribute */
        UINT    maxmsz ;    /* Maximum message size (in bytes) */
        SIZE    mbfsz ;     /* Size of message buffer area (in bytes) */
        VP      mbf ;       /* Start sddress of message buffer area */
        /* Other implementation specific fields may be added. */
} T_CMBF ;

typedef struct t_rmbf {
```

```
        ID      stskid ;    /* ID number of the task at the head of the
                               message buffer's send-wait queue */
        ID      rtskid ;    /* ID number of the task at the head of the
                               message buffer's receive-wait queue */
        UINT    smsgcnt ;   /* The number of messages in the message
                               buffer */
        SIZE    fmbfsz ;    /* Size of free message buffer area in bytes,
                               without the minimum control areas */
        /* Other implementation specific fields may be added. */
} T_RMBF ;
```

The following represents the function codes for the message buffer service calls:

TFN_CRE_MBF	–0x89	Function code of cre_mbf
TFN_ACRE_MBF	–0xc7	Function code of acre_mbf
TFN_DEL_MBF	–0x8a	Function code of del_mbf
TFN_SND_MBF	–0x8d	Function code of snd_mbf
TFN_PSND_MBF	–0x8e	Function code of psnd_mbf
TFN_TSND_MBF	–0x8f	Function code of tsnd_mbf
TFN_RCV_MBF	–0x91	Function code of rcv_mbf
TFN_PRCV_MBF	–0x92	Function code of prcv_mbf
TFN_TRCV_MBF	–0x93	Function code of trcv_mbf
TFN_REF_MBF	–0x94	Function code of ref_mbf

[Supplemental Information]

Figure 4-2 shows the behavior of a message buffer when the size of the message buffer area is 0. In this figure, task A and task B are assumed to be running asynchronously.

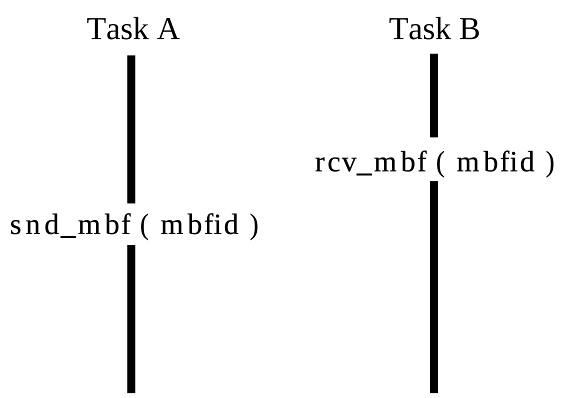


Figure 4-2. Synchronous Communication through a Message Buffer

- If task A invokes snd_mbf first, task A is moved to the WAITING state until task B invokes rcv_mbf. During this time, task A is in the sending waiting state for a message buffer.
- If, on the other hand, task B invokes rcv_mbf first, task B is moved to the WAITING state until task A invokes snd_mbf. During this time, task B is in the receiv-

ing waiting state for a message buffer.

- When task A invokes snd_mbf and task B invokes rcv_mbf, the message transfer from task A and task B takes place. After this, both tasks are moved to the runnable state.

Tasks that are waiting to send a message to a message buffer will send their messages in the order that the tasks are placed in the wait queue. An example is when task A tries to send a 40 byte message to a message buffer, and task B tries to send a 10 byte message to the same message buffer. Assume that these tasks are placed in the wait queue so that task A is ahead of task B. A third task then receives a message 20 byte long, resulting in 20 bytes of available area in the message buffer. Even though task B only needs 10 bytes to send its message, it cannot do so until task A has sent its message. However, an implementation-specific extension can add an attribute to the message buffer that will allow task B to send its message before task A in this example.

A message buffer transfers a variable-sized message through copying. It is different from a data queue in that it transfers variable-sized messages. It is different from a mailbox in that it copies the messages.

A message buffer is assumed to be implemented as a ring buffer.

If a message buffer is used for the kernel's error log (for recording errors that cannot be reported to the processing unit that invoked a service call), a message buffer with an ID number of (–4) can be used. Furthermore, message buffers with ID numbers (–3) and (–2) can be used when message buffers are used inside the kernel to communicate with debug support functions. Limiting the access to these message buffers from application programs is also allowed.

[Differences from the µITRON3.0 Specification]

Whether tasks should send messages according to their order in the wait queue or according to which task can send a message first was implementation-dependent in the µITRON3.0 Specification. The µITRON4.0 Specifications has determined the former order to be standard.

CRE_MBF	Create Message Buffer (Static API)
cre_mbf	Create Message Buffer
acre_mbf	Create Message Buffer (ID Number Automatic Assignment)

[Static API]

```
CRE_MBF ( ID mbfid, { ATR mbfatr, UINT maxmsz, SIZE mbfsz,
                      VP mbf } ) ;
```

[C Language API]

```
ER ercd = cre_mbf ( ID mbfid, T_CMBF *pk_cmbf ) ;
ER_ID mbfid = acre_mbf ( T_CMBF *pk_cmbf ) ;
```

[Parameter]

ID	mbfid	ID number of the message buffer to be created (except acre_mbf)
T_CMBF *	pk_cmbf	Pointer to the packet containing the message buffer creation information (in CRE_MBF, packet contents must be directly specified.)

pk_cmbf includes (T_CMBF type)

ATR	mbfatr	Message buffer attribute
UINT	maxmsz	Maximum message size (in bytes)
SIZE	mbfsz	Size of message buffer area (in bytes)
VP	mbf	Start address of message buffer area

(Other implementation specific information may be added.)

[Return Parameter]

cre_mbf:

ER	ercd	E_OK for normal completion or error code

acre_mbf:

ER_ID	mbfid	ID number (positive value) of the created message buffer or error code

[Error Code]

E_ID	Invalid ID number (mbfid is invalid or unusable; only cre_mbf)
E_NOID	No ID number available (there is no message buffer ID assignable; only acre_mbf)
E_NOMEM	Insufficient memory (message buffer area cannot be allocated)
E_RSATR	Reserved attribute (mbfatr is invalid or unusable)
E_PAR	Parameter error (pk_cmbf, maxmsz, mbfsz, or mbf is invalid)
E_OBJ	Object state error (message buffer is already registered; only

cre_mbf)

[Functional Description]

These service calls create a message buffer with an ID number specified by mbfid based on the information contained in the packet pointed to by pk_cmbf. mbfatr is the attribute of the message buffer. maxmsz is the maximum size in bytes of the message that can be sent to the message buffer. mbfsz is the size of the message buffer area in bytes. mbf is the start address of the message buffer area.

In CRE_MBF, mbfid is an integer parameter with automatic assignment. mbfatr is a preprocessor constant expression parameter.

acre_mbf assigns a message buffer ID from the pool of unassigned message buffer IDs and returns the assigned message buffer ID.

mbfatr can be specified as (TA_TFIFO || TA_TPRI). If TA_TFIFO (= 0x00) is specified, the message buffer's send-wait queue will be in FIFO order. If TA_TPRI (= 0x01) is specified, the message buffer's send-wait queue will be in task priority order.

The memory area starting from mbf and whose size is mbfsz is used as the message buffer area. Because the information for message management is also placed in the message buffer area, the whole message buffer area cannot be used to store messages. An application program can estimate the size to be specified in mbfsz by using the TSZ_MBF macro. If mbf is NULL (= 0), the kernel allocates the necessary memory area in bytes specified by mbfsz. mbfsz may be specified as 0.

When maxmsz is specified as 0, an E_PAR error is returned.

[Supplemental Information]

The message buffer's receive-wait queue always utilizes the FIFO ordering. Also, the messages in a message buffer is always in FIFO order.

[Differences from the µITRON3.0 Specification]

In µITRON3.0, the TA_TPRI attribute of a message buffer indicated that the receive-wait queue is priority-ordered. In µITRON4.0, it has changed to indicate that the send-wait queue is priority-ordered. This is because the priority-ordered send-wait queue is more effective than priority-ordered receive-wait queue.

The start address of the message buffer area (mbf) has been added to the message buffer creation information. The extended information has been removed. The parameter name has been changed from bufsz to mbfsz and the order of maxmsz and mbfsz in the creation information packet has been exchanged. The data type of maxmsz has been changed from INT to UINT and that of mbfsz has been changed from INT to SIZE.

acre_mbf has been newly added.

del_mbf Delete Message Buffer

[C Language API]

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

[Parameter]

ID	mbfid	ID number of the message buffer to be deleted

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (mbfid is invalid or unusable)
E_NOEXS	Non-existent object (specified message buffer is not registered)

[Functional Description]

This service call deletes the message buffer specified by mbfid. If the message buffer area was allocated by the kernel, the area is released.

[Supplemental Information]

The messages in the message buffer will be discarded. See Section 3.8 for information regarding handling tasks that are waiting in the message buffer's send-wait queue and receive-wait queue when the message buffer is deleted.

snd_mbf	Send to Message buffer
psnd_mbf	Send to Message buffer (Polling)
tsnd_mbf	Send to Message buffer (with Timeout)

[C Language API]

```
ER ercd = snd_mbf ( ID mbfid, VP msg, UINT msgsz ) ;
ER ercd = psnd_mbf ( ID mbfid, VP msg, UINT msgsz ) ;
ER ercd = tsnd_mbf ( ID mbfid, VP msg, UINT msgsz,
                                  TMO tmout ) ;
```

[Parameter]

ID	mbfid	ID number of the message buffer to which the message is sent
VP	msg	Start address of the message to be sent
UINT	msgsz	Size of the message to be sent (in bytes)
TMO	tmout	Specified timeout (only tsnd_mbf)

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (mbfid is invalid or unusable)
E_NOEXS	Non-existent object (specified message buffer is not registered)
E_PAR	Parameter error (msg, msgsz, tmout is invalid)
E_RLWAI	Forced release from waiting (accept rel_wai while waiting; except psnd_mbf)
E_TMOUT	Polling failure or timeout (except snd_mbf)
E_DLT	Waiting object deleted (message buffer is deleted while waiting; except psnd_mbf)

[Functional Description]

These service calls send a message to the message buffer specified by mbfid. The message to be sent is placed in the memory area starting from the address specified by msg and its size in bytes is specified by msgsz. Specifically, the following actions are performed.

If there are already tasks in the message buffer's receive-wait queue, the task at the head of the receive-wait queue is selected to receive the message. These service calls copy the sent message to the memory area specified by the task for receiving a message and release the task from waiting. The released task receives the size of the sent message (msgsz) as the return value of the service call that caused it to wait in the receive-wait queue.

If no tasks are waiting in the message buffer's receive-wait queue, the behavior of these service calls depends on whether there is a task already waiting to send its message before the invoking task. These service calls will copy the sent message to the tail of the message buffer if either: 1) no task is waiting to send a message to the specified message buffer, or 2) the message buffer has the TA_TPRI (= 0x01) attribute set and the priorities of the other tasks that are waiting to send messages are lower than the invoking task. If neither of these conditions is satisfied, or if there is no room in the message buffer area to store the sent message, the invoking task is placed in the send-wait queue and is moved to the sending waiting state for the message buffer.

If there are already tasks in the message buffer's send-wait queue, the invoking task is placed in the send-wait queue as described below. When the message buffer's attribute has TA_TFIFO (= 0x00) set, the invoking task is placed at the tail of the send-wait queue. When the message buffer's attribute has TA_TPRI (= 0x01) set, the invoking task is placed in the send-wait queue in the order of the task's priority. If the send-wait queue contains tasks with the same priority as the invoking task, the invoking task is placed after those tasks.

When the first task in the send-wait queue has changed as the result of releasing a task in the wait queue from waiting with rel_wai, ter_tsk, or a timeout, the actions, when possible, to make the tasks send messages starting from the new first task in the wait queue are necessary. Since the specific actions are similar to the actions to be taken after rcv_mbf has removed a message from the message buffer, see the functional description of rcv_mbf for more details. The same actions are also necessary when the first task in the send-wait queue has changed as the result of changing the priority of a task in the wait queue by chg_pri or mutex operations.

psnd_mbf is a polling service call with the same functionality as snd_mbf. tsnd_mbf has the same functionality as snd_mbf with an additional timeout feature. tmout can be set to a positive number indicating a timeout duration or it can be set to TMO_POL (= 0) or TMO_FEVR (= –1).

When msgsz is larger than the maximum message size of the message buffer, an E_PAR error is returned. An E_PAR error is also returned when msgsz is 0.

[Supplemental Information]

tsnd_mbf acts the same as psnd_mbf if TMO_POL is specified in tmout as long as no context error occurs. Also, tsnd_mbf acts the same as snd_mbf if TMO_FEVR is specified in tmout.

[Differences from the μITRON3.0 Specification]

The order of the parameters has been changed. The data type of msgsz has been changed from INT to UINT.

rcv_mbf	Receive from Message Buffer
prcv_mbf	Receive from Message Buffer (Polling)
trcv_mbf	Receive from Message Buffer (with Timeout)

[C Language API]

```
ER_UINT msgsz = rcv_mbf ( ID mbfid, VP msg ) ;
ER_UINT msgsz = prcv_mbf ( ID mbfid, VP msg ) ;
ER_UINT msgsz = trcv_mbf ( ID mbfid, VP msg, TMO tmout ) ;
```

[Parameter]

ID	mbfid	ID number of the message buffer from which a message is received
VP	msg	Start address of the memory area to store the received message
TMO	tmout	Specified timeout (only trcv_mbf)

[Return Parameter]

ER_UINT	msgsz	Size of the received message (in byte, positive value) or error code

[Error Code]

E_ID	Invalid ID number (mbfid is invalid or unusable)
E_NOEXS	Non-existent object (specified message buffer is not registered)
E_PAR	Parameter error (msg or tmout is invalid)
E_RLWAI	Forced release from waiting (accept rel_wai while waiting; except prcv_mbf)
E_TMOUT	Polling failure or timeout (except rcv_mbf)
E_DLT	Waiting object deleted (message buffer is deleted while waiting; except prcv_mbf)

[Functional Description]

These service calls receive a message from the message buffer specified by mbfid and stores it in the memory area starting from the address specified by msg. The size of the received message in bytes is returned through msgsz. Specifically, the following actions are performed.

If the message buffer already has messages, these service calls copy the first message to the memory area starting from the address specified by msg and return the message size through msgsz. The copied message is deleted from the message buffer area. If there are tasks in the message buffer's send-wait queue, the service calls check if there is enough room for the message of the task at the head of the wait queue after deleting the received message. If there is enough room, the message of the task at the head of the wait queue is copied to the tail of the message buffer and the task is released from waiting. The released task receives E_OK from the service call that caused it to wait in the wait queue. When some tasks still remain in the send-wait queue after the release of the task, the same actions must be repeated on the new head task in the wait queue.

If there are no messages in the message buffer and if there are tasks in the message buffer's send-wait queue (this occurs when the size of the message buffer area is too small for the message of the task at the head of the wait queue), the message from the task at the head of the send-wait queue is copied to the memory area starting from the address specified by msg. The size of the copied message is returned through msgsz. The task is released from waiting and receives E_OK from the service call that caused it to wait in the send-wait queue.

If there are no messages in the message buffer and if there are no tasks in the send-wait queue, the invoking task is placed in the receive-wait queue and moved to the receiving waiting state for the message buffer. If there are already tasks in the receive-wait queue, the invoking task is placed at the tail of the receive-wait queue.

prcv_mbf is a polling service call with the same functionality as rcv_mbf. trcv_mbf has the same functionality as rcv_mbf with an additional timeout feature. tmout can be set to a positive number indicating a timeout duration or it can be set to TMO_POL (0) or TMO_FEVR (–1).

[Supplemental Information]

If these service calls release more than one task from waiting, the order of release corresponds with the order in which the tasks are placed in the wait queue. Therefore, among the same priority tasks moved to the runnable state, the task closer to the head of the wait queue has higher precedence.

trcv_mbf acts the same as prcv_mbf if TMO_POL is specified in tmout as long as no context error occurs. Also, trcv_mbf acts the same as rcv_mbf if TMO_FEVR is specified in tmout.

[Differences from the μITRON3.0 Specification]

The size of the received message (msgsz) is now returned as the return value of the service calls. The order of parameters has been changed. The data type of msgsz has been changed from INT to UINT (the actual type though is ER_UINT).

ref_mbf Reference Message Buffer State

[C Language API]

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

[Parameter]

ID	mbfid	ID number of the message buffer to be referenced
T_RMBF *	pk_rmbf	Pointer to the packet returning the message buffer state

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

pk_rmbf includes (T_RMBF type)

ID	stskid	ID number of the task at the head of the send-wait queue
ID	rtskid	ID number of the task at the head of the receive-wait queue
UINT	smsgcnt	The number of messages in the message buffer
SIZE	fmbfsz	Size of free message buffer area in bytes, without the minimum control areas

(Other implementation specific information may be added.)

[Error Code]

E_ID	Invalid ID number (mbfid is invalid or unusable)
E_NOEXS	Non-existent object (specified message buffer is not registered)
E_PAR	Parameter error (pk_rmbf is invalid)

[Functional Description]

This service call references the state of the message buffer specified by mbfid. The state of the message buffer is returned through the packet pointed to by pk_rmbf.

The ID number of the task at the head of the message buffer's send-wait queue is returned through stskid. If no tasks are waiting to send a message, TSK_NONE (= 0) is returned instead.

The ID number of the task at the head of the message buffer's receive-wait queue is returned through rtskid. If no tasks are waiting to receive a message, TSK_NONE (= 0) is returned instead.

The number of messages currently in the message buffer is returned through smsgcnt.

The size of the minimum control area subtracted from the size of the free message buffer area in bytes is returned through fmbfsz. Specifically, fmbfsz is the maximum message size that can be stored in the free message buffer area when there is not enough room for a message with the maximum message size. If the message buffer has enough room to store a message with the maximum message size, fmbfsz is the approximate total size of messages that can be stored in the free message buffer area.

[Supplemental Information]

A message with smaller size than fmbfsz may not always be sent at once without entering the WAITING state. This happens if there are tasks already waiting to send a message to the message buffer (when stskid ≠ TSK_NONE).

A message buffer cannot have tskid ≠ TSK_NONE and smsgcnt ≠ 0 at the same time. When stskid ≠ TSK_NONE, fmbfsz is smaller than the maximum message size.

[Differences from the μITRON3.0 Specification]

The extended information has been removed from the reference information. The ID number of the task at the head of each wait queue is now returned instead of a boolean value indicating whether a task is waiting or not. The number of messages in the message buffer is now returned instead of the size of the message to be received next. Based on these replacements, the names and data types of the return parameters have been changed. The size of the minimum control area is excluded from the size returned through fmbfsz in order to make the returned value strictly standardized to the message size when the free message buffer area is small.

The name of the return parameter frbufsz has been changed to fmbfsz and its data type has been changed from INT to SIZE. The order of parameters and of return parameters has been changed.

4.5.3 Rendezvous

The rendezvous feature is used for synchronization and communication between tasks. It supports a procedure to handle a processing request from one task to another task and the return of the result to the requesting task. The object used to coordinate this task interaction is called a rendezvous port. The rendezvous feature is typically used to realize a client/server model communication, but it also provides a more flexible synchronous communication model.

Rendezvous functions include the ability to create and delete a rendezvous port, to request a processing at a rendezvous port (calling rendezvous), to accept a processing request at a rendezvous port (accepting rendezvous), to return a processed result (terminating rendezvous), to forward a processing request to another rendezvous port (forwarding rendezvous), and to reference the state of a rendezvous port and of a rendezvous. A rendezvous port is an object identified with an ID number. The ID number of a rendezvous port is called the rendezvous port ID.

A task which requests a processing at a rendezvous port (the client task) calls for a rendezvous by specifying a rendezvous port, a rendezvous condition, and a message that contains information about the requested processing. The message is referred to as the calling message. A task that receives a processing request (the server task) accepts the rendezvous by specifying the rendezvous port and the rendezvous condition.

A rendezvous condition is specified by a bit pattern. A rendezvous is only established when the bit patterns of the rendezvous conditions of both the calling task and the accepting task match. The match is performed by taking the logical AND of the corresponding bits. If the result is not 0, the rendezvous is established. The calling task will be in the calling waiting state for the rendezvous until the rendezvous is established. On the other hand, the accepting task will be in the accepting waiting state for the rendezvous until the rendezvous is established.

When a rendezvous is established, the calling message is transferred from the calling task to the accepting task. The calling task is moved to the termination waiting state for the rendezvous and waits for the processing to be completed. The accepting task is released from the accepting waiting state for the rendezvous and executes the requested processing. Once the accepting task completes its processing, it returns the result to the calling task as a return message, and the rendezvous is terminated. At this time, the calling task is released from the termination waiting state for the rendezvous.

A rendezvous port has an associated call-wait queue to hold the tasks in the calling waiting state for a rendezvous and an accept-wait queue to hold the tasks in the accepting waiting state for a rendezvous. Once a rendezvous is established, the two tasks are detached from the rendezvous port. A rendezvous port does not have a wait queue to hold the tasks that are in the termination waiting state for a rendezvous. Also, it does not have information about the two tasks involved with the requested processing.

The kernel assigns an object number to a rendezvous in order to distinguish multiple rendezvous. The object number of a rendezvous is called the rendezvous number. The process for assigning rendezvous numbers is implementation-dependent. However, the rendezvous number should at least include information regarding the task that called the rendezvous. Each rendezvous should have a unique rendezvous number if possible. For example, if the same task calls a rendezvous port twice, the first and second rendezvous should have different rendezvous numbers.

The following data types are used for rendezvous functions:

RDVPTN	Bit pattern of the rendezvous condition (unsigned integer)
RDVNO	Rendezvous number

The following kernel configuration constant is defined for use with rendezvous functions:

TBIT_RDVPTN	The number of bits in a rendezvous condition (the number of bits of RDVPTN type)

The following data type packets are defined for creating and referencing rendezvous ports and rendezvous:

```
typedef struct t_cpor {
        ATR     poratr ;    /* Rendezvous port attribute */
        UINT    maxcmsz ;   /* Maximum calling message size (in
                               bytes) */
        UINT    maxrmsz ;   /* Maximum return message size (in
                               bytes) */
        /* Other implementation specific fields may be added. */
} T_CPOR ;

typedef struct t_rpor {
        ID      ctskid ;    /* ID number of the task at the head of the
                               rendezvous port's call-wait queue */
        ID      atskid ;    /* ID number of the task at the head of the
                               rendezvous port's accept-wait queue */
        /* Other implementation specific fields may be added. */
} T_RPOR ;

typedef struct t_rrdv {
        ID      wtskid ;    /* ID number of the task in the termination
                               waiting state for the rendezvous */
        /* Other implementation specific fields may be added. */
} T_RRDV ;
```

The following represents the function codes for rendezvous service calls:

TFN_CRE_POR	–0x95	Function code of cre_por
TFN_ACRE_POR	–0xc8	Function code of acre_por
TFN_DEL_POR	–0x96	Function code of del_por
TFN_CAL_POR	–0x97	Function code of cal_por
TFN_TCAL_POR	–0x98	Function code of tcal_por

TFN_ACP_POR	–0x99	Function code of acp_por
TFN_PACP_POR	–0x9a	Function code of pacp_por
TFN_TACP_POR	–0x9b	Function code of tacp_por
TFN_FWD_RDV	–0x9c	Function code of fwd_rdv
TFN_RPL_RDV	–0x9d	Function code of rpl_rdv
TFN_REF_POR	–0x9e	Function code of ref_por
TFN_REF_RDV	–0x9f	Function code of ref_rdv

[Supplemental Information]

A rendezvous is a synchronization and communication function which was introduced by the ADA language specification and is based on CSP (Communicating Sequential Processes). However, the ADA rendezvous is a part of the language specification and its premise is different from the µITRON4.0 Specification rendezvous. In particular, the rendezvous offered by a real-time kernel is intended to be a primitive for realizing the language rendezvous. There are several differences between the ADA rendezvous and the µITRON4.0 Specification rendezvous. Because of this, the rendezvous port of the µITRON4.0 Specification cannot always be used in realizing the ADA rendezvous.

Figure 4-3 shows the behavior of a rendezvous. In this figure, task A and task B are assumed to be running asynchronously.

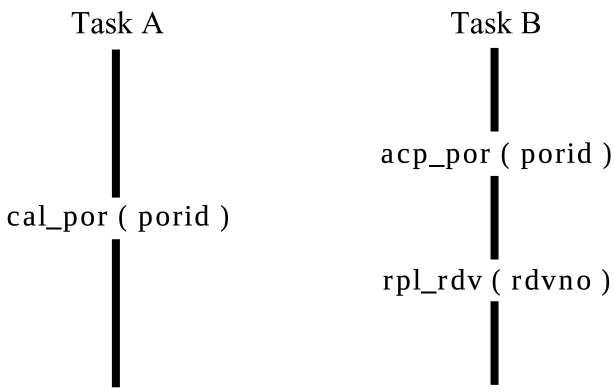


Figure 4-3. Rendezvous Operation

- If task A invokes cal_por first, task A is moved to the WAITING state until task B invokes acp_por. During this time, task A is in the calling waiting state for the rendezvous.
- If, on the other hands, task B invokes acp_por first, task B is moved to the WAITING state until task A invokes cal_por. During this time, task B is in the accepting waiting state for the rendezvous.
- When task A invokes cal_por and task B invokes acp_por, the rendezvous is established. When this happens, task B is released from waiting while task A remains in the WAITING state. Task A, at this time, is in the termination waiting state for the rendezvous.

- Once task B invokes rpl_rdv, task A is released from waiting. Both tasks are moved to the runnable state.

One example of assigning a rendezvous number is to use the ID number of the task that called the rendezvous as the lower bits, and then assign a serial number to the remaining upper bits. So if the task ID is a 16-bit value, the rendezvous number should be made 32 bits by adding a 16-bit serial value.

[Differences from the μITRON3.0 Specification]

The term rendezvous port is now used instead of port.

The data type of the parameter that contains the rendezvous condition bit pattern has been changed from UINT to the new data type RDVPTN. The data type for a rendezvous number has been changed from RNO to RDVNO.

[Rationale]

Although a rendezvous feature can be realized by combining other synchronization and communication features, writing application programs involving return messages with rendezvous functions is much easier and more efficient. For example, a rendezvous does not need an area to store messages because the two tasks wait until the message transfer is completed.

When a task calls a rendezvous port multiple times, each rendezvous number must be unique if possible for the following reason. Assume that a task is in the termination waiting state for a rendezvous and that the task is released from waiting due to timeout or forced release. After being released, if it calls a rendezvous port again that is successfully established, the rendezvous numbers of the previous and the current rendezvous would be the same. When another task tries to terminate the previous rendezvous, the current one would be terminated by mistake if they have the same number. By assigning two different numbers to two different rendezvous and by recording the rendezvous number with the waiting task, an error can be detected when the first rendezvous is terminated.

CRE_POR	Create Rendezvous Port (Static API)
cre_por	Create Rendezvous Port
acre_por	Create Rendezvous Port (ID Number Automatic Assignment)

[Static API]

CRE_POR (ID porid, { ATR poratr, UINT maxcmsz,
UINT maxrmsz }) ;

[C Language API]

ER ercd = cre_por (ID porid, T_CPOR *pk_cpor) ;
ER_ID porid = acre_por (T_CPOR *pk_cpor) ;

[Parameter]

ID	porid	ID number of the rendezvous port to be created (except acre_por)
T_CPOR *	pk_cpor	Pointer to the packet containing the rendezvous port creation information (in CRE_POR, packet contents must be directly specified.)

pk_cpor includes (T_CPOR type)

ATR	poratr	Rendezvous port attribute
UINT	maxcmsz	Maximum calling message size (in bytes)
UINT	maxrmsz	Maximum return message size (in bytes)

(Other implementation specific information may be added.)

[Return Parameter]

cre_por:

ER	ercd	E_OK for normal completion or error code

acre_por:

ER_ID	porid	ID number (positive value) of the created rendezvous port or error code

[Error Code]

E_ID	Invalid ID number (porid is invalid or unusable; only cre_por)
E_NOID	No ID number available (there is no rendezvous port ID assignable; only acre_por)
E_RSATR	Reserved attribute (poratr is invalid or unusable)
E_PAR	Parameter error (pk_cpor, maxcmsz, or maxrmsz is invalid)
E_OBJ	Object state error (specified rendezvous port is already registered; only cre_por)

[Functional Description]

These service calls create a rendezvous port with an ID number specified by porid based on the information contained in the packet pointed to by pk_cpor. poratr is the rendezvous port attribute. maxcmsz is the maximum size in bytes of a calling message. maxrmsz is the maximum size in bytes of a returned message.

In CRE_POR, porid is an integer parameter with automatic assignment. poratr is a preprocessor constant expression parameter.

acre_por assigns a rendezvous port ID from the pool of unassigned rendezvous port IDs and returns the assigned rendezvous port ID.

poratr can be specified as (TA_TFIFO || TA_TPRI). If TA_TFIFO (= 0x00) is specified, the rendezvous port's call-wait queue will be in FIFO order. If TA_TPRI(= 0x01) is specified, the rendezvous port's call-wait queue will be in task priority order.

maxcmsz and maxrmsz may be specified as 0.

[Supplemental Information]

The rendezvous port's accept-wait queue always utilizes FIFO ordering.

[Differences from the μITRON3.0 Specification]

By specifying the TA_TRPI attribute, a rendezvous port's call-wait queue will now be in task priority order.

The extended information has been removed from the rendezvous port creation information. The data types of maxcmsz and maxrmsz have been changed from INT to UINT.

acre_por has been newly added.

del_por Delete Rendezvous Port

[C Language API]

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

[Parameter]

ID	porid	ID number of the rendezvous port to be deleted

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (porid is invalid or unusable)
E_NOEXS	Non-existent object (specified rendezvous port is not registered)

[Functional Description]

This service call deletes the rendezvous port specified by porid.

[Supplemental Information]

Deleting a rendezvous port does not affect an already established rendezvous. The deletion is not reported to a task that has accepted a rendezvous and is already executing the requested processing. The task that called the rendezvous and is in the termination waiting state for the rendezvous will still continue waiting. Moreover, a termination of the rendezvous is executed normally even if the rendezvous port is already deleted.

See Section 3.8 for information regarding handling tasks that are waiting to call or accept a rendezvous at the rendezvous port when the rendezvous port is deleted.

cal_por Call Rendezvous
tcal_por Call Rendezvous (with Timeout)

[C Language API]

```
ER_UINT rmsgsz = cal_por ( ID porid, RDVPTN calptn, VP msg,
                           UINT cmsgsz ) ;
ER_UINT rmsgsz = tcal_por ( ID porid, RDVPTN calptn, VP msg,
                           UINT cmsgsz, TMO tmout ) ;
```

[Parameter]

ID	porid	ID number of the rendezvous port to be called
RDVPTN	calptn	Bit pattern of the rendezvous condition at the calling side
VP	msg	Start address of the calling message and of the memory area to store the return message
UINT	cmsgsz	Calling message size (in bytes)
TMO	tmout	Specified timeout (only tcal_por)

[Return Parameter]

ER_UINT	rmsgsz	Return message size (in bytes, positive value or 0) or error code

[Error Code]

E_ID	Invalid ID number (porid is invalid or unusable)
E_NOEXS	Non-existent object (specified rendezvous port is not registered)
E_PAR	Parameter error (calptn, msg, cmsgsz, or tmout is invalid)
E_RLWAI	Forced release from waiting (accept rel_wai while waiting)
E_TMOUT	Polling failure or timeout (only tcal_por)
E_DLT	Waiting object deleted (rendezvous port is deleted while waiting)

[Functional Description]

These service calls call for a rendezvous at the port specified by porid with the rendezvous condition specified by calptn. The start address of the calling message is specified by msg and its size in bytes is specified by cmsgsz. The service calls store the return message in the memory area starting from msg and return its size in bytes through rmsgsz. Specifically, the following actions are performed.

If there is a task in the accepting waiting state for the rendezvous at the rendezvous port, these service calls establish a rendezvous if the rendezvous conditions of the invoking task and the waiting task match. If there are more than one task in the accepting waiting state for the rendezvous, these service calls check their rendezvous conditions one by one starting from the task at the head of the accept-wait queue. The service calls establish a rendezvous with the first task that matches the rendezvous condition.

When a rendezvous is established, these service calls assign a rendezvous number to the established rendezvous and move the invoking task to the termination waiting state for the rendezvous. The service calls also copy the calling message into the memory area specified by the accepting task, which was in the accepting waiting state for the rendezvous. The service calls then release the task from waiting. The released task receives the calling message size (cmsgsz) as the return value of the service call that caused it to wait in the accept-wait queue and the assigned rendezvous number through rdvno.

If no tasks are waiting to accept a rendezvous at the specified rendezvous port, or if none of the waiting tasks has a matching rendezvous condition, the invoking task is placed in the call-wait queue and is moved to the calling waiting state for the rendezvous.

If there are already tasks in the rendezvous port's call-wait queue, the invoking task is placed in the call-wait queue as described below. When the rendezvous port's attribute has TA_TFIFO (= 0x00) set, the invoking task is placed at the tail of the call-wait queue. When rendezvous port's attribute has TA_TPRI (= 0x01) set, the invoking task is placed in the call-wait queue in the order of the task's priority. If the call-wait queue contains tasks with the same priority as the invoking task, the invoking tasks is placed after those tasks.

tcal_por has same functionality as cal_por with an additional timeout feature. If the rendezvous does not terminate after a period specified by tmout starting from when tcal_por is called, tcal_por returns an E_TMOUT error. tmout can be set to TMO_FEVR (= –1) in addition to a positive number indicating a timeout duration. When TMO_POL (= 0) is specified, an E_PAR error is returned.

If tcal_por is invoked and results in a timeout after it establishes a rendezvous, the status of the rendezvous cannot be recovered to its former state before it was established. This is an exception to the rule stating that "side effects due to a service call that returns an error code do not arise." In this case, an error is reported to the accepting task when the task tries to terminate the rendezvous. This also applies to the case where a task is forcibly released from the termination waiting state for the rendezvous with rel_wai. In this case, the service call returns an E_RLWAI error. On the contrary, since deleting a rendezvous port does not affect an already established rendezvous, the service call never returns an E_DLT error once the rendezvous is established.

An E_PAR error is returned when calptn is 0 or when cmsgsz exceeds the maximum calling message size. cmsgsz may be specified as 0.

[Supplemental Information]

When there is a possibility that a rendezvous might be forwarded, the application should allocate enough memory area, starting from the address specified by msg, to store a return message with the maximum size regardless of the expected return message size. The application should also assume that the contents of the allocated memory area will be destroyed. This is because when the rendezvous is forwarded, the calling message may be copied to the memory area starting from the address specified by msg.

tcal_por acts the same as cal_por if TMO_FEVR is specified in tmout.

[Differences from the μITRON3.0 Specification]

The interpretation of timeout in tcal_por has been changed. As a result, pcal_por became unnecessary and is removed from the μITRON4.0 Specification. tcal_por returns an E_PAR error if TMO_POL is specified in tmout.

A calling message with a size of 0 is now allowed.

The return message size (rmsgsz) is now returned as the return value of the service calls. The data type of calptn has been changed from UINT to RDVPTN. The data types of cmsgsz and rmsgsz have been changed from INT to UINT (the actual type though is ER_UINT for rmsgsz). The order of parameters and of return parameters has been changed.

[Rationale]

The reason an E_PAR error is returned when 0 is specified for calptn is that a rendezvous is never established in this case, which in turn would never release the invoking task from calling waiting state for the rendezvous.

acp_por	Accept Rendezvous
pacp_por	Accept Rendezvous (Polling)
tacp_por	Accept Rendezvous (with Timeout)

[C Language API]

```
ER_UINT cmsgsz = acp_por ( ID porid, RDVPTN acpptn,
                           RDVNO *p_rdvno, VP msg ) ;
ER_UINT cmsgsz = pacp_por ( ID porid, RDVPTN acpptn,
                           RDVNO *p_rdvno, VP msg ) ;
ER_UINT cmsgsz = tacp_por ( ID porid, RDVPTN acpptn,
                           RDVNO *p_rdvno, VP msg, TMO tmout ) ;
```

[Parameter]

ID	porid	ID number of the rendezvous port where a rendezvous is accepted
RDVPTN	acpptn	Bit pattern of the rendezvous condition at the accepting side
VP	msg	Start address of the memory area to store the calling message
TMO	tmout	Specified timeout (only tacp_por)

[Return Parameter]

ER_UINT	cmsgsz	Calling message size (in bytes, positive value or 0) or error code
RDVNO	rdvno	Rendezvous number of the established rendezvous

[Error Code]

E_ID	Invalid ID number (porid is invalid or unusable)
E_NOEXS	Non-existent object (specified rendezvous port is not registered)
E_PAR	Parameter error (acpptn, msg, or tmout is invalid)
E_RLWAI	Forced release from waiting (accept rel_wai while waiting; except pacp_por)
E_TMOUT	Polling failure or timeout (except acp_por)
E_DLT	Waiting object deleted (rendezvous port is deleted while waiting; except pacp_por)

[Functional Description]

These service calls accept a rendezvous at the rendezvous port specified by porid with the rendezvous condition specified by acpptn. The calling message is stored in the memory area starting from the address specified by msg and its size in bytes is returned through cmsgsz. The rendezvous number of the established rendezvous is returned through rdvno. Specifically, the following actions are performed.

If there is a task in the calling waiting state for the rendezvous at the rendezvous port, these service calls establish a rendezvous if the rendezvous conditions of the invoking task and the waiting task match. If there are more than one task in the calling waiting state for the rendezvous, these service calls check their rendezvous conditions one by one starting from the task at the head of the call-wait queue. The service calls establish a rendezvous with the first task that matches the rendezvous condition.

When a rendezvous is established, these service calls assign a rendezvous number to the established rendezvous and return the rendezvous number through rdvno. The service calls also copy the calling message of the calling task, which was in the calling waiting state for the rendezvous, to the memory area starting from the address specified by msg and return the calling message size through cmsgsz. The task is then removed from the rendezvous port's call-wait queue and is moved to the termination waiting state for the rendezvous.

If no tasks are waiting to call a rendezvous at the specified rendezvous port, or if none of the waiting tasks has a matching rendezvous condition, the invoking task is placed in the accept-wait queue and is moved to the accepting waiting state for the rendezvous. If there are already tasks in the accept-wait queue, the invoking task is placed at the tail of the accept-wait queue.

pacp_por is a polling service call with the same functionality as acp_por. tacp_por has the same functionality as acp_por with an additional timeout feature. tmout can be set to a positive number indicating a timeout duration or it can be set to TMO_POL (= 0) or TMO_FEVR (= –1).

An E_PAR error is returned when acpptn is 0.

[Supplemental Information]

A task that has established a rendezvous with another task with acp_por may accept a rendezvous again with acp_por before the previous rendezvous has been terminated. The new rendezvous can be accepted at either the same rendezvous port as the previously established one or at another rendezvous port. If the same rendezvous port is used, the task can have multiple established rendezvous at the same rendezvous port. Furthermore, the calling task of the previously established rendezvous can be released from waiting either by timeout or forced release. When the task calls the rendezvous again, the task can have multiple rendezvous with the other task at the same rendezvous port.

tacp_por acts the same as pacp_por if TMO_POL is specified in tmout as long as no context error occurs. Also, tacp_por acts the same as acp_por if TMO_FEVR is specified in tmout.

[Differences from the μITRON3.0 Specification]

The calling message size (cmsgsz) is now returned as the return value of the service calls. The data type of acpptn has been changed from UINT to RDVPTN. The data type of rdvno has been changed from RNO to RDVNO. The data type of cmsgsz has been changed from INT to UINT (the actual type though is ER_UINT). The order of parameters and of return parameters has been changed.

[Rationale]

The reason an E_PAR error is returned when 0 is specified for acpptn is that a rendezvous is never established in this case, which in turn would never release the invoking task from accepting waiting state for the rendezvous.

fwd_por Forward Rendezvous

[C Language API]

```
ER ercd = fwd_por ( ID porid, RDVPTN calptn, RDVNO rdvno,
                    VP msg, UINT cmsgsz ) ;
```

[Parameter]

ID	porid	ID number of the rendezvous port to which the rendezvous is forwarded
RDVPTN	calptn	Bit pattern of the rendezvous condition at the calling side
RDVNO	rdvno	Rendezvous number to be forwarded
VP	msg	Start address of the calling message
UINT	cmsgsz	Calling message size (in bytes)

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (porid is invalid or unusable)
E_NOEXS	Non-existent object (specified rendezvous port is not registered)
E_PAR	Parameter error (calptn, msg, or cmsgsz is invalid)
E_ILUSE	Illegal service call use (maximum return message size of the rendezvous port to which the rendezvous is forwarded is too large)
E_OBJ	Object state error (rdvno is invalid)

[Functional Description]

This service call forwards the rendezvous specified by rdvno with the rendezvous condition specified by calptn to the rendezvous port specified by porid. The start address of the calling message after forwarding is specified by msg and its size in bytes is specified by cmsgsz.

When fwd_por is invoked, the result is the same as if the task that called the rendezvous specified by rdvno (called task A below) has called the rendezvous port specified by portid with the rendezvous condition calptn and the calling message msg.

The operations of fwd_por is described in detail as follows.

If a task is waiting to accept a rendezvous at the rendezvous port to which the rendezvous is forwarded, and if the rendezvous condition of the waiting task and that specified by calptn match, this service call establishes a rendezvous between the task and task A. If there are more than one task waiting to accept a rendezvous, this service call check their rendezvous conditions one by one starting from the task at the head of the accept-wait queue. The service call establishes a rendezvous with the first task that matches the rendezvous condition.

When a rendezvous is established, this service call assigns a rendezvous number to the established rendezvous and moves task A to the termination waiting state for the rendezvous. The service call also copies the calling message specified by msg and cmsgsz into the memory area specified by the accepting task, which was in the accepting waiting state for the rendezvous. The service call then releases the task from waiting. The released task receives the calling message size (cmsgsz) as the return value of the service call that caused it to wait in the accept-wait queue and the assigned rendezvous number through rdvno.

If no tasks are waiting to accept a rendezvous at the rendezvous port to which the rendezvous is forwarded, or if none of the waiting tasks has a matching rendezvous condition, task A is placed in the call-wait queue of the rendezvous port to which the rendezvous is forwarded, and is moved to the calling waiting state for the rendezvous. The calling message specified by msg and cmsgsz is copied to the memory area specified by task A to store the return message.

If there are already tasks in the rendezvous port's call-wait queue, task A is placed in the call-wait queue as described below. If the rendezvous port's attribute has TA_TFIFO (= 0x00) set, task A is placed at the tail of the call-wait queue. If the rendezvous port's attribute has TA_TPRI (= 0x01) set, task A is placed in the call-wait queue in the order of the task's priority. If the call-wait queue contains tasks with the same priority as task A, task A is placed after those tasks.

The maximum return message size of the rendezvous port to which the rendezvous is forwarded must be smaller than or equal to that of the rendezvous port at which the rendezvous was established. Otherwise an E_ILUSE error is returned.

When cmsgsz is larger than the maximum calling message size of the rendezvous port to which the rendezvous is forwarded, or when cmsgsz is larger than the return message size of the rendezvous port at which the rendezvous was established, an E_PAR error is returned. cmsgsz may be specified as 0.

A rendezvous number accepted by another task may also be specified in rdvno. In other words, the task that invokes fwd_por and forwards the rendezvous does not necessarily correspond to the task that has accepted the rendezvous.

If the task that has called the rendezvous specified by rdvno is not in the termination waiting state for the same rendezvous, an E_OBJ error is returned. An E_OBJ error is also returned when the value specified by rdvno cannot be interpreted as a rendezvous number.

An E_PAR error is returned when calptn is 0.

[Supplemental Information]

Since the result of invoking fwd_por is the same as if task A has called the rendezvous port, the record of forwarding a rendezvous is not necessary. For this reason, a forwarded rendezvous can be forwarded again.

Since the execution of fwd_por ends immediately, the task that invokes fwd_por never enters the WAITING state. The application can reuse the area in which the calling message was stored for other purposes after fwd_por returns because the calling message specified by msg and cmsgsz is copied to another area during the execution of fwd_por. After fwd_por returns, the task that invoked fwd_por is detached from the following: the rendezvous port at which the rendezvous was established, the rendezvous port to which the rendezvous is forwarded, the forwarded rendezvous, and the newly established rendezvous if any.

A timeout specified for tcal_por applies to the interval from the invocation of tcal_por to the termination of the rendezvous. Therefore, if task A called a rendezvous by tcal_por, the specified timeout continues to be valid after the rendezvous is forwarded.

The rendezvous port to which the rendezvous is forwarded may be the same rendezvous port at which the rendezvous was originally established. In this case, the accepted rendezvous is returned to the original state before it was established. However, the rendezvous pattern and the calling message are changed to those specified for fwd_por.

Even if the task that has called the rendezvous is released from the termination waiting state for the rendezvous due to a timeout or a forced release after the rendezvous is established, its release would not be notified to the task that has accepted the rendezvous. In this case, an E_OBJ error is returned if the task that accepted the rendezvous invokes fwd_por and tries to forward the rendezvous. The task can determine whether the calling task for the rendezvous is still in the termination waiting state by invoking ref_rdv.

Figure 4-4 illustrates a server distribution task using fwd_por.

[Differences from the μITRON3.0 Specification]

When task A is moved to the calling waiting state for a rendezvous, the calling message specified by msg and cmsgsz is now defined to be stored in the area in which task A stores the return message.

The handling of timeout in fwd_por has been changed according to the changed interpretation of timeout for tcal_por.

The fact that a task other than the task that has accepted the rendezvous can forward the rendezvous is now clarified.

The calling message size can now be specified as 0.

The data types of calptn, rdvno, and cmsgsz have been changed from UINT to RDVPTN, from RNO to RDVNO, and from INT to UINT, respectively.

[Rationale]

In order to reduce the number of states the system should handle, the specification does

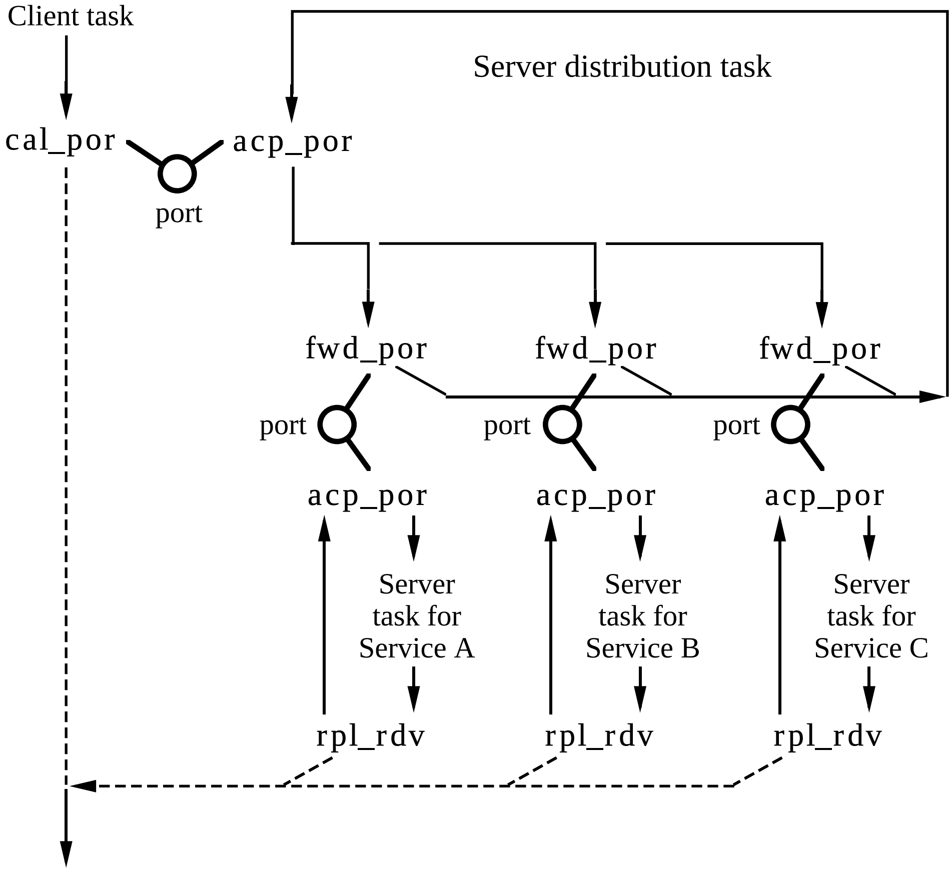


Figure 4-4. Server Distribution Task using fwd_por

not require the record of forwarding a rendezvous. In cases where the record is necessary, the rendezvous may be called, instead of forwarded by fwd_por, using nested cal_por.

The following states the reason why an error is returned when the maximum return message size of the rendezvous port to which the rendezvous is forwarded is larger than that of the rendezvous port at which the rendezvous was established. Task A must allocate a memory area that can hold a return message of the maximum allowed size from the rendezvous port that task A first called. If the maximum return message size of the rendezvous port to which the rendezvous is forwarded is larger, the return message may not fit in the allocated area.

An error is returned if cmsgsz is larger than the maximum return message size of the rendezvous port at which the rendezvous was established. This is because when task A is moved to the calling waiting state for a rendezvous, task A copies the calling message specified by msg and cmsgsz to the area it allocated for storing the return message.

rpl_rdv Terminate Rendezvous

[C Language API]

```
ER ercd = rpl_rdv ( RDVNO rdvno, VP msg, UINT rmsgsz ) ;
```

[Parameter]

RDVNO	rdvno	Rendezvous number to be terminated
VP	msg	Start address of the return message
UINT	rmsgsz	Return message size (in bytes)

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_PAR	Parameter error (msg or rmsgsz is invalid)
E_OBJ	Object state error (rdvno is invalid)

[Functional Description]

This service call terminates the rendezvous specified by rdvno. The start address of the return message is specified by msg and its size in bytes is specified by rmsgsz.

Specifically, if the task which has called the rendezvous specified by rdvno is in the termination waiting state for the rendezvous, this service call copies the return message specified by msg and rmsgsz to the area allocated by the calling task to store the return message. The service call then releases the task from waiting. The released task receives the return message size (rmsgsz) as the return value of the service call that caused it to wait.

If the task that has called the rendezvous specified by rdvno is not in the termination waiting state for the same rendezvous, an E_OBJ error is returned. An E_OBJ error is also returned when the value specified by rdvno cannot be interpreted as a rendezvous number.

A rendezvous number accepted by another task may also be specified in rdvno. In other words, the task that invokes rpl_rdv and terminates the rendezvous does not necessarily correspond to the task that has accepted the rendezvous.

When rmsgsz is larger than the maximum return message size of the rendezvous port to which the rendezvous was established, an E_PAR error is returned. rmsgsz may be specified as 0.

[Supplemental Information]

Even if the task that has called the rendezvous is released from the termination waiting state for the rendezvous due to a timeout or a forced release after the rendezvous is established, its release would not be notified to the task that has accepted the rendezvous. In this case, an E_OBJ error is returned if the task that accepted the rendezvous invokes rpl_rdv and tries to terminate the rendezvous. The task can determine whether the calling task for the rendezvous is still in the termination waiting state by invoking ref_rdv.

After the rendezvous is established, both the calling and accepting tasks are detached from the rendezvous port. However, the maximum return message size for the rendezvous port is necessary for checking if the return message size (rmsgsz) is smaller than or equal to the maximum size. For this reason, the maximum return message size must be saved in conjunction with the rendezvous. The maximum size, for example, can be stored in the TCB of the task in the calling waiting state or in an area (such as the stack area) that can be referenced from the TCB.

[Differences from the μITRON3.0 Specification]

The fact that a task other than the task that has accepted the rendezvous can terminate the rendezvous is now clarified.

The return message size can now be specified as 0.

The data types of rdvno and rmsgsz have been changed from RNO to RDVNO and from INT to UINT, respectively.

[Rationale]

A rendezvous port ID is not passed as a parameter to rpl_rdv because the task that has called the rendezvous is detached from the rendezvous port once the rendezvous is established.

When rdvno is invalid, an E_OBJ error is returned instead of an E_PAR error. This is because an invalid value of rdvno cannot be detected statically.

ref_por Reference Rendezvous Port State

[C Language API]

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

[Parameter]

ID	porid	ID number of the rendezvous port to be referenced
T_RPOR *	pk_rpor	Pointer to the packet returning the rendezvous port state

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

pk_rpor includes (T_RPOR type)

ID	ctskid	ID number of the task at the head of the call-wait queue
ID	atskid	ID number of the task at the head of the accept-wait queue

(Other implementation specific information may be added.)

[Error Code]

E_ID	Invalid ID number (porid is invalid or unusable)
E_NOEXS	Non-existent object (specified rendezvous port is not registered)
E_PAR	Parameter error (pk_rpor is invalid)

[Functional Description]

This service call references the state of the rendezvous port specified by porid. The state of the rendezvous port is returned through the packet pointed to by pk_rpor.

The ID number of the task at the head of the rendezvous port's call-wait queue is returned through ctskid. If no tasks are waiting to call a rendezvous at the rendezvous port, TSK_NONE (= 0) is returned instead.

The ID number of the task at the head of the rendezvous port's accept-wait queue is returned through atskid. If no tasks are waiting to accept a rendezvous at the rendezvous port, TSK_NONE (= 0) is returned instead.

[Differences from the µITRON3.0 Specification]

The extended information has been removed from the reference information. The ID number of the task at the head of each wait queue is now returned instead of a boolean value indicating whether a task is waiting or not. Based on this replacement, the names and data types of the return parameters have been changed. The order of parameters and of return parameters has been changed.

ref_rdv Reference Rendezvous State

[C Language API]

ER ercd = ref_rdv (RDVNO rdvno, T_RRDV *pk_rrdv) ;

[Parameter]

RDVNO	rdvno	Rendezvous number of the rendezvous to be referenced
T_RRDV *	pk_rrdv	Pointer to the packet returning the rendezvous state

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

pk_rrdv includes (T_RRDV type)

ID	wtskid	ID number of the task in the termination waiting state for the rendezvous

(Other implementation specific information may be added.)

[Error Code]

E_PAR	Parameter error (pk_rrdv is invalid)

[Functional Description]

This service call references the state of the rendezvous to which the rendezvous number specified by rdvno is assigned. The state of the rendezvous is returned through the packet pointed to by pk_rrdv.

When the task that has called the rendezvous specified by rdvno is in the termination waiting state for the same rendezvous, the ID number of the task is returned through wtskid. If the task is not in the termination waiting state for the same rendezvous, or if the rdvno cannot be interpreted as a rendezvous number, TSK_NONE (= 0) is returned instead.

[Supplemental Information]

When this service call invoked with a rendezvous number returns a task ID through wtskid, rpl_rdv or fwd_por invoked with the same rendezvous number never returns an E_OBJ error.

[Differences from the μITRON3.0 Specification]

ref_rdv has been newly added. The ITRON2 Specification had a corresponding service call, rdv_sts.

4.6 Memory Pool Management Functions

Memory pool management functions provide dynamic memory management by software. Memory pool management functions include fixed-sized memory pool and variable-sized memory pool.

[Supplemental Information]

The µITRON4.0 Specification does not specify functions for multiple logical memory spaces or hardware memory management unit (MMU).

4.6.1 Fixed-Sized Memory Pools

A fixed-sized memory pool is an object for dynamically managing fixed-sized memory blocks. The fixed-sized memory pool functions include the ability to create and delete a fixed-sized memory pool, to acquire and release a memory block to/from a fixed-sized memory pool, and to reference the state of a fixed-sized memory pool. A fixed-sized memory pool is an object identified by an ID number. The ID number of a fixed-sized memory pool is called the fixed-sized memory pool ID.

A fixed-sized memory pool has an associated memory area where fixed-sized memory blocks are allocated (this is called fixed-sized memory pool area or simply memory pool area) and an associated wait queue for acquiring a memory block. If there are no memory blocks available, a task trying to acquire a memory block from the fixed-sized memory pool will be in the waiting state for a fixed-sized memory block until a memory block is released. The task waiting to acquire a fixed-sized memory block is placed in the fixed-sized memory pool's wait queue.

The following kernel configuration macro is defined for use with the fixed-sized memory pool functions:

SIZE mpfsz = TSZ_MPF (UINT blkcnt, UINT blksz)

This macro returns the total required size of the fixed-size memory pool area in bytes necessary to allocate blkcnt memory blocks each of size blksz bytes.

The following data type packets are defined for creating and referencing fixed-sized memory pools:

```
typedef struct t_cmpf {
        ATR     mpfatr ;    /* Fixed-sized memory pool attribute */
        UINT    blkcnt ;    /* Total number of memory blocks */
        UINT    blksz ;     /* Memory block size (in bytes) */
        VP      mpf ;       /* Start address of the fixed-sized memory
                               pool area */
        /* Other implementation specific fields may be added. */
} T_CMPF ;

typedef struct t_rmpf {
```

```
        ID          wtskid ;    /* ID number of the task at the head of the
                                   fixed-sized memory pool's wait
                                   queue */
        UINT        fblkcnt ;   /* Number of free memory blocks in the
                                   fixed-sized memory pool */
        /* Other implementation specific fields may be added. */
} T_RMPF ;
```

The following represents the functions codes for the fixed-sized memory pool service calls:

TFN_CRE_MPF	–0x45	Function code of cre_mpf
TFN_ACRE_MPF	–0xc9	Function code of acre_mpf
TFN_DEL_MPF	–0x46	Function code of del_mpf
TFN_GET_MPF	–0x49	Function code of get_mpf
TFN_PGET_MPF	–0x4a	Function code of pget_mpf
TFN_TGET_MPF	–0x4b	Function code of tget_mpf
TFN_REL_MPF	–0x47	Function code of rel_mpf
TFN_REF_MPF	–0x4c	Function code of ref_mpf

[Standard Profile]

The Standard Profile requires support for fixed-sized memory pool functions except for dynamic creation and deletion of a fixed-sized memory pool (cre_mpf, acre_mpf, del_mpf) and reference of a fixed-sized memory pool state (ref_mpf).

The Standard Profile does not require TSZ_MPF to be defined.

[Supplemental Information]

When using fixed-sized memory pool functions for memory blocks of different sizes, a fixed-sized memory pool should be created for each size.

CRE_MPF	Create Fixed-Sized Memory Pool (Static API)	[S]
cre_mpf	Create Fixed-Sized Memory Pool	
acre_mpf	Create Fixed-Sized Memory Pool (ID Number Automatic Assignment)	

[Static API]

CRE_MPF (ID mpfid, { ATR mpfatr, UINT blkcnt, UINT blksz, VP mpf }) ;

[C Language API]

ER ercd = cre_mpf (ID mpfid, T_CMPF *pk_cmpf) ;
ER_ID mpfid = acre_mpf (T_CMPF *pk_cmpf) ;

[Parameter]

ID	mpfid	ID number of the fixed-sized memory pool to be created (except acre_mpf)
T_CMPF *	pk_cmpf	Pointer to the packet containing the fixed-sized memory pool creation information (in CRE_MPF, packet contents must be directly specified.)

pk_cmpf includes (T_CMPF type)

ATR	mpfatr	Fixed-sized memory pool attribute
UINT	blkcnt	Total number of memory blocks
UINT	blksz	Memory block size (in bytes)
VP	mpf	Start address of the fixed-sized memory pool area

(Other implementation specific information may be added.)

[Return Parameter]

cre_mpf:

ER	ercd	E_OK for normal completion or error code

acre_mpf:

ER_ID	mpfid	ID number (positive value) of the created fixed-sized memory pool or error code

[Error Code]

E_ID	Invalid ID number (mpfid is invalid or unusable; only cre_mpf)
E_NOID	No ID number available (there is no fixed-sized memory pool ID assignable; only acre_mpf)
E_NOMEM	Insufficient memory (memory pool area cannot be allocated)
E_RSATR	Reserved attribute (mpfatr is invalid or unusable)
E_PAR	Parameter error (pk_cmpf, blkcnt, blksz, or mpf is invalid)
E_OBJ	Object state error (specified fixed-sized memory pool is

already registered; only cre_mpf)

[Functional Description]

These service calls create a fixed-sized memory pool with an ID number specified by mpfid based on the information contained in the packet pointed to by pk_cmpf. mpfatr is the attribute of the fixed-sized memory pool. blkcnt is the total number of memory blocks. blksz is size of each memory block. mpf is the start address of the fixed-sized memory pool area.

In CRE_MPF, mpfid is an integer parameter with automatic assignment. mpfatr is a preprocessor constant expression parameter.

acre_mpf assigns a fixed-sized memory pool ID from the pool of unassigned fixed-sized memory pool IDs and returns the assigned fixed-sized memory pool ID.

mpfatr can be specified as (TA_TFIFO || TA_TPRI). If TA_TFIFO (= 0x00) is specified, the fixed-sized memory pool's wait queue will be in FIFO order. If TA_TPRI (= 0x00) is specified, the fixed-sized memory pool's wait queue will be in task priority order.

The necessary area to hold up to blkcnt memory blocks, each of size blksz bytes, starts from mpf and is used as the fixed-size memory pool area. An application program can calculate the size of the memory pool area necessary to hold blkcnt number of memory blocks, each of size blksz bytes, by using the TSZ_MPF macro. If mpf is NULL (= 0), the kernel allocates the necessary memory area. When blkcnt or blksz is specified as 0, an E_PAR error is returned.

[Standard Profile]

The Standard Profile does not require support for when other values than NULL is specified in mpf.

[Differences from the µITRON3.0 Specification]

The start address of the memory pool area (mpf) has been added to the fixed-sized memory pool creation information. The extended information has been removed. The names of the parameters have been changed from mpfcnt to blkcnt and from blfsz to blksz, respectively. In addition, their data types have been changed from INT to UINT.

acre_mpf has been newly added.

del_mpf Delete Fixed-Sized Memory Pool

[C Language API]

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

[Parameter]

ID	mpfid	ID number of the fixed-sized memory pool to be deleted

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (mpfid is invalid or unusable)
E_NOEXS	Non-existent object (specified fixed-sized memory pool is not registered)

[Functional Description]

This service call deletes the fixed-sized memory pool specified by mpfid. If the memory pool area was allocated by the kernel, the area is released.

[Supplemental Information]

See Section 3.8 for information regarding handling tasks that are waiting for a memory block from the fixed-sized memory pool when the fixed-sized memory pool is deleted.

get_mpf	Acquire Fixed-Sized Memory Block	[S]
pget_mpf	Acquire Fixed-Sized Memory Block (Polling)	[S]
tget_mpf	Acquire Fixed-Sized Memory Block (with Timeout)	[S]

[C Language API]

```
ER ercd = get_mpf ( ID mpfid, VP *p_blk ) ;
ER ercd = pget_mpf ( ID mpfid, VP *p_blk ) ;
ER ercd = tget_mpf ( ID mpfid, VP *p_blk, TMO tmout ) ;
```

[Parameter]

ID	mpfid	ID number or the fixed-sized memory pool from which a memory block is acquired
TMO	tmout	Specified timeout (only tget_mpf)

[Return Parameter]

ER	ercd	E_OK for normal completion or error code
VP	blk	Start address of the acquired memory block

[Error Code]

E_ID	Invalid ID number (mpfid is invalid or unusable)
E_NOEXS	Non-existent object (specified fixed-sized memory pool is not registered)
E_PAR	Parameter error (p_blk or tmout is invalid)
E_RLWAI	Forced release from waiting (accept rel_wai while waiting; except pget_mpf)
E_TMOUT	Polling failure or timeout (except get_mpf)
E_DLT	Waiting object deleted (fixed-sized memory pool is deleted while waiting; except pget_mpf)

[Functional Description]

These service calls acquire a memory block from the fixed-sized memory pool specified by mpfid. The size of the memory block is specified during the creation of the fixed-sized memory pool. The start address of the memory block is returned through blk. Specifically, when free memory blocks are available in the memory pool area, one of the memory blocks is selected and takes on an acquired status. If there are no memory blocks available, the invoking task is placed in the fixed-sized memory pool's wait queue and is moved to the waiting state for a fixed-sized memory block.

If there are already tasks in the fixed-sized memory pool's wait queue, the invoking task is placed in the wait queue as described below. When the fixed-sized memory pool's attribute has TA_TFIFO (= 0x00) set, the invoking task is placed at the tail of the wait queue. When the fixed-sized memory pool's attribute has TA_TPRI (= 0x01) set, the invoking task is placed in the wait queue in the order of the task's priority. If the wait queue contains tasks with the same priority as the invoking tasks, the invoking task is placed after those tasks.

pget_mpf is a polling service call with the same funcionality as get_mpf. tget_mpf has the same functionality as get_mpf with an additional timeout feature. tmout can be set to a positive number indicating a timeout duration or it can be set to TMO_POL (= 0) or TMO_FEVR (= –1).

[Supplemental Information]

The size of the acquired memory block may be larger than the memory block size that was specified during the creation of the fixed-sized memory pool. Since these service calls do not clear the memory block, its contents are undefined.

tget_mpf acts the same as pget_mpf if TMO_POL is specified in tmout as long as no context error occurs. Also, tget_mpf acts the same as get_mpf if TMO_FEVR is specified in tmout.

[Differences from the μITRON3.0 Specification]

The names of the service calls have been changed from get_blf, pget_blf, and tget_blf to get_mpf, pget_mpf, and tget_mpf, respectively. The order of parameters and of return parameters has been changed.

rel_mpf Release Fixed-Sized Memory Block [S]

[C Language API]

```
ER ercd = rel_mpf ( ID mpfid, VP blk ) ;
```

[Parameter]

ID	mpfid	ID number of the fixed-sized memory pool to which the memory block is released
VP	blk	Start address of the memory block to be released

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (mpfid is invalid or unusable)
E_NOEXS	Non-existent object (specified fixed-sized memory pool is not registered)
E_PAR	Parameter error (blk is invalid, release to a different memory pool, specified address is not the start address of a memory block)

[Functional Description]

This service call releases the memory block starting from the address specified by blk to the fixed-sized memory pool specified by mpfid.

If there are already tasks in the fixed-sized memory pool's wait queue, this service call lets the task at the head of the wait queue acquire the released memory block and releases the task from waiting. The released task receives E_OK from the service call that caused it to wait in the fixed-sized memory pool's wait queue. It also receives the value specified by blk as the start address of the acquired memory block.

The fixed-sized memory pool to which the memory block is released must be the same fixed-sized memory pool from which the memory block was acquired. Otherwise, an E_PAR error is returned.

The start address of the memory block to be released must be the start address of an acquired memory block returned by get_mpf, pget_mpf, or tget_mpf. In addition, the memory block must not be a released memory block. The behavior is undefined when other addresses are specified in blk. When an error should be reported, an E_PAR error is returned.

[Differences from the µITRON3.0 Specification]

The name of the service call has been changed from rel_blf to rel_mpf. The name of the parameter has been changed from blf to blk.

ref_mpf Reference Fixed-Sized Memory Pool State

[C Language API]

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

[Parameter]

ID	mpfid	ID number of the fixed-sized memory pool to be referenced
T_RMPF *	pk_rmpf	Pointer to the packet returning the fixed-sized memory pool state

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

pk_rmpf includes (T_RMPF type)

ID	wtskid	ID number of the task at the head of the wait queue
UINT	fblkcnt	Number of free memory blocks

(Other implementation specific information may be added.)

[Error Code]

E_ID	Invalid ID number (mpfid is invalid or unusable)
E_NOEXS	Non-existent object (specified fixed-sized memory pool is not registered)
E_PAR	Parameter error (pk_rmpf is invalid)

[Functional Description]

This service call references the state of the fixed-sized memory pool specified by mpfid. The state of the fixed-sized memory pool is returned through the packet pointed to by pk_rmpf.

The ID number of the task at the head of the fixed-sized memory pool's wait queue is returned through wtskid. If no tasks are waiting to acquire a memory block, TSK_NONE (= 0) is returned instead.

The number of free memory blocks in the fixed-sized memory pool area is returned through fblkcnt.

[Supplemental Information]

A fixed-sized memory pool cannot have wtskid ≠ TSK_NONE and fblkcnt ≠ 0 at the same time.

[Differences from the μITRON3.0 Specification]

The extended information has been removed from the reference information. The ID number of the task at the head of the wait queue is now returned instead of a boolean value indicating whether a task is waiting or not. Based on this replacement, the name and data type of the return parameter has been changed.

The name of the return parameter has been changed from frbcnt to fblkcnt, and its data type has been changed from INT to UNIT. The order of parameters and of return parameters has been changed.

4.6.2 Variable-Sized Memory Pools

A variable-sized memory pool is an object for dynamically managing variable-sized memory blocks. The variable-sized memory pool functions include the ability to create and delete a variable-sized memory pool, to acquire and release a memory block to/from a variable-sized memory pool, and to reference the state of a variable-sized memory pool. A variable-sized memory pool is an object identified by an ID number. The ID number of a variable-sized memory pool is called the variable-sized memory ID.

A variable-sized memory pool has an associated memory area where variable-sized memory blocks are allocated (this is called variable-sized memory pool area or simply memory pool area) and an associated wait queue for acquiring a memory block. If there are no memory blocks available, a task trying to acquire a memory block from the variable-sized memory pool will be in the waiting state for a variable-sized memory block until enough memory blocks are released. The task waiting to acquire a variable-sized memory block is placed in the variable-sized memory pool's wait queue.

The following kernel configuration macro is defined for use with variable-sized memory pool functions:

> SIZE mplsz = TSZ_MPL (UINT blkcnt, UINT blksz)
>
> This macro returns an approximate size in bytes necessary to allocate blkcnt memory blocks each of size blksz bytes.

This macro is only an estimation for determining the size of the memory pool area. It cannot be used to determine the total required size of a memory pool area to allocate memory blocks with different sizes. In addition, when the memory pool area becomes fragmented, the specified number of memory blocks cannot be allocated.

The following data type packets are defined for creating and referencing variable-sized memory pools:

```
typedef struct t_cmpl {
    ATR     mplatr ;    /* Variable-sized memory pool attribute */
    SIZE    mplsz ;     /* Size of the variable-sized memory pool
                           area (in bytes) */
    VP      mpl ;       /* Start address of the variable-sized
                           memory pool area */
    /* Other implementation specific fields may be added. */
} T_CMPL ;

typedef struct t_rmpl {
    ID      wtskid ;    /* ID number of the task at the head of the
                           variable-sized memory pool's wait
                           queue */
    SIZE    fmplsz ;    /* Total size of free memory blocks in the
                           variable-sized memory pool (in
                           bytes) */
```

```
        UINT        fblksz ;      /* Maximum memory block size available
                                     (in bytes) */
        /* Other implementation specific fields may be added. */
} T_RMPL ;
```

The following represents the functions codes for the variable-sized memory pool service calls:

TFN_CRE_MPL	–0xa1	Function code of cre_mpl
TFN_ACRE_MPL	–0xca	Function code of acre_mpl
TFN_DEL_MPL	–0xa2	Function code of del_mpl
TFN_GET_MPL	–0xa5	Function code of get_mpl
TFN_PGET_MPL	–0xa6	Function code of pget_mpl
TFN_TGET_MPL	–0xa7	Function code of tget_mpl
TFN_REL_MPL	–0xa3	Function code of rel_mpl
TFN_REF_MPL	–0xa8	Function code of ref_mpl

[Standard Profile]

The Standard Profile does not require support for variable-sized memory pool functions.

[Supplemental Information]

Tasks that are waiting for a memory block from a variable-sized memory pool will acquire a memory block in the order that the tasks are placed in the wait queue. An example is when task A tries to acquire 400 byte memory block from a variable-sized memory pool and task B tries to acquire 100 byte memory block from the same variable-sized memory pool. Assume that these tasks are placed in the wait queue so that task A is ahead of task B. A third task then releases 200 byte memory block to the variable-sized memory pool, resulting in 200 bytes of available area in the variable-sized memory pool. Even though task B only needs 100 bytes to acquire a memory block, it cannot do so until task A has acquired a memory block. However, an implementation-specific extension can add an attribute to the variable-sized memory pool that will allow task B to acquire a memory block before task A in this example.

[Differences from the µITRON3.0 Specification]

Whether tasks should acquire memory blocks according to their order in the wait queue or according to which task can acquire a memory block first was implementation-dependent in the µITRON3.0 Specification. The µITRON4.0 Specifications has determined the former order to be standard.

CRE_MPL Create Variable-Sized Memory Pool (Static API)
cre_mpl Create Variable-Sized Memory Pool
acre_mpl Create Variable-Sized Memory Pool (ID Number Automatic Assignment)

[Static API]

CRE_MPL (ID mplid, { ATR mplatr, SIZE mplsz, VP mpl }) ;

[C Language API]

ER ercd = cre_mpl (ID mplid, T_CMPL *pk_cmpl) ;

ER_ID mplid = acre_mpl (T_CMPL *pk_cmpl) ;

[Parameter]

ID	mplid	ID number of the variable-sized memory pool to be created (except acre_mpl)
T_CMPL *	pk_cmpl	Pointer to the packet containing the variable-sized memory pool creation information (in CRE_MPL, packet contents must be directly specified.)

pk_cmpl includes (T_CMPL type)

ATR	mplatr	Variable-sized memory pool attribute
SIZE	mplsz	Size of the variable-sized memory pool area (in bytes)
VP	mpl	Start address of the variable-sized memory pool area

(Other implementation specific information may be added.)

[Return Parameter]

cre_mpl:

ER	ercd	E_OK for normal completion or error code

acre_mpl:

ER_ID	mplid	ID number (positive value) of the created variable-sized memory pool or error code

[Error Code]

E_ID	Invalid ID number (mplid is invalid or unusable; only cre_mpl)
E_NOID	No ID number available (there is no variable-sized memory pool ID assignable; only acre_mpl)
E_NOMEM	Insufficient memory (memory pool area cannot be allocated)
E_RSATR	Reserved attribute (mplatr is invalid or unusable)
E_PAR	Parameter error (pk_cmpl, mplsz, or mpl is invalid)
E_OBJ	Object state error (specified variable-sized memory pool is already registered; only cre_mpl)

[Functional Description]

These service calls create a variable-sized memory pool with an ID number specified by mplid based on the information contained in the packet pointed to by pk_cmpl. mplatr is the attribute of the variable-sized memory pool. mplsz is the size of the variable-sized memory pool area in bytes. mpl is the start address of the variable-sized memory pool area.

In CRE_MPL, mplid is an integer parameter with automatic assignment. mplatr is a preprocessor constant expression parameter.

acre_mpl assigns a variable-sized memory pool ID from the pool of unassigned variable-sized memory pool IDs and returns the assigned variable-sized memory pool ID.

mplatr can be specified as (TA_TFIFO || TA_TPRI). If TA_TFIFO (= 0x00) is specified, the variable-sized memory pool's wait queue will be in FIFO order. If TA_TPRI (= 0x00) is specified, the variable-sized memory pool's wait queue will be in task priority order.

The memory area starting from mpl and whose size is mplsz is used as the memory pool area. Because the information for memory block management is also placed in the memory pool area, the whole memory pool area cannot be used to allocate memory blocks. An application program can estimate the size to be specified in mplsz by using the TSZ_MPL macro. If mpl is NULL (= 0), the kernel allocates the necessary memory area in bytes specified by mplsz. When mplsz is specified as 0, an E_PAR error is returned.

[Differences from the μITRON3.0 Specification]

The start address of the memory pool area (mpl) has been added to the variable-sized memory pool creation information. The extended information has been removed. The data type of mplsz has been changed from INT to SIZE.

acre_mpl has been newly added.

del_mpl Delete Variable-Sized Memory Pool

[C Language API]

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

[Parameter]

ID	mplid	ID number of the variable-sized memory pool to be deleted

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (mplid is invalid or unusable)
E_NOEXS	Non-existent object (specified variable-sized memory pool is not registered)

[Functional Description]

This service call deletes the variable-sized memory pool specified by mplid. If the memory pool area was allocated by the kernel, the area is released.

[Supplemental Information]

See Section 3.8 for information regarding handling tasks that are waiting for a memory block from the variable-sized memory pool when the variable-sized memory pool is deleted.

get_mpl Acquire Variable-Sized Memory Block
pget_mpl Acquire Variable-Sized Memory Block (Polling)
tget_mpl Acquire Variable-Sized Memory Block (with Timeout)

[C Language API]

```
ER ercd = get_mpl ( ID mplid, UINT blksz, VP *p_blk ) ;
ER ercd = pget_mpl ( ID mplid, UINT blksz, VP *p_blk ) ;
ER ercd = tget_mpl ( ID mplid, UINT blksz, VP *p_blk,
                     TMO tmout ) ;
```

[Parameter]

ID	mplid	ID number of the variable-sized memory pool from which a memory block is acquired
UINT	blksz	Memory block size to be acquired (in bytes)
TMO	tmout	Specified timeout (only tget_mpl)

[Return Parameter]

ER	ercd	E_OK for normal completion or error code
VP	blk	Start address of the acquired memory block

[Error Code]

E_ID	Invalid ID number (mplid is invalid or unusable)
E_NOEXS	Non-existent object (specified variable-sized memory pool is not registered)
E_PAR	Parameter error (p_blk, tmout is invalid)
E_RLWAI	Forced release from waiting (accept rel_wai while waiting; except pget_mpl)
E_TMOUT	Polling failure or timeout (except get_mpl)
E_DLT	Waiting object deleted (variable-sized memory pool is deleted while waiting; except pget_mpl)

[Functional Description]

These service calls acquire a memory block whose size is specified by blksz from the variable-sized memory pool specified by mplid. The start address of the memory block is returned through blk.

Specific actions to be performed depend on whether there is a task waiting to acquire a memory block with precedence over the invoking task. If no tasks are waiting to acquire a memory block from the variable-sized memory block, or if the variable-sized memory pool's attribute has TA_TPRI (= 0x01) set and the invoking task has higher priority than all of the waiting tasks, a memory block of size blksz bytes is acquired from the memory pool area. If the conditions are not satisfied or if the free memory area is insufficient for acquiring a memory block, the invoking task is placed in the variable-sized memory pool's wait queue and is moved to the waiting state for a variable-sized memory block.

If there are already tasks in the variable-sized memory pool's wait queue, the invoking task is placed in the wait queue as described below. When the variable-sized memory pool's attribute has TA_TFIFO (= 0x00) set, the invoking task is placed at the tail of the wait queue. When the variable-sized memory pool's attribute has TA_TPRI (= 0x01) set, the invoking task is placed in the wait queue in the order of the task's priority. If the wait queue contains tasks with the same priority as the invoking tasks, the invoking task is placed after those tasks.

When the first task in the wait queue has changed as the result of releasing a task in the wait queue from waiting with rel_wai, ter_tsk, or a timeout, the actions, when possible, to make the tasks acquire memory blocks starting from the new first task in the wait queue are necessary. Since the specific actions are similar to the actions to be taken after rel_mpl has released a memory block to the variable-sized memory pool, see the functional description of rel_mpl for more details. The same actions are also necessary when the first task in the wait queue has changed as the result of changing the priority of a task in the wait queue by chg_pri or mutex operations.

pget_mpl is a polling service call with the same funcionality as get_mpl. tget_mpl has the same functionality as get_mpl with an additional timeout feature. tmout can be set to a positive number indicating a timeout duration or it can be set to TMO_POL (= 0) or TMO_FEVR (= –1).

[Supplemental Information]

The size of the acquired memory block may be larger than the size specified by blksz. Since these service calls do not clear the memory block, its contents are undefined.

tget_mpl acts the same as pget_mpl if TMO_POL is specified in tmout as long as no context error occurs. Also, tget_mpl acts the same as get_mpl if TMO_FEVR is specified in tmout.

[Differences from the μITRON3.0 Specification]

The names of the service calls have been changed from get_blk, pget_blk, and tget_blk to get_mpl, pget_mpl, and tget_mpl, respectively. The data type of blksz has been changed from INT to UINT. The order of parameters and of return parameters has been changed.

rel_mpl Release Variable-Sized Memory Block

[C Language API]

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

[Parameter]

ID	mplid	ID number of the variable-sized memory pool to which the memory block is released
VP	blk	Start address of memory block to be released

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (mplid is invalid or unusable)
E_NOEXS	Non-existent object (specified variable-sized memory pool is not registered)
E_PAR	Parameter error (blk is invalid, release to a different memory pool, specified address is not the start address of a memory block)

[Functional Description]

This service call release the memory block starting from the address specified by blk to the variable-sized memory pool specified by mplid.

If there are already tasks in the variable-sized memory pool's wait queue, this service call checks if, as a result of releasing the memory block, the first task in the wait queue can acquire a memory block of the requested size. If the requested size is met, the service call lets the task acquire the memory block and releases the task from waiting. The released task receives E_OK from the service call that caused it to wait in the variable-sized memory pool's wait queue. It also receives the start address of the acquired memory block. When some tasks still remain in the wait queue after the release of the task, the same actions must be repeated on the new head task in the wait queue.

The variable-sized memory pool to which the memory block is released must be the same variable-sized memory pool from which the memory block was acquired. Otherwise, an E_PAR error is returned.

The start address of the memory block to be released must be the start address of an acquired memory block returned by get_mpl, pget_mpl, or tget_mpl. In addition, the memory block must not be a released memory block. The behavior is undefined when other addresses are specified in blk. When an error should be reported, an E_PAR error is returned.

[Supplemental Information]

If this service call releases more than one task from waiting, the order of release corresponds with the order in which the tasks are placed in the wait queue. Therefore, among the same priority tasks moved to the runnable state, the task closer to the head of the wait queue has higher precedence.

[Differences from the μITRON3.0 Specification]

The name of the service call has been changed from rel_blk to rel_mpl.

ref_mpl Reference Variable-Sized Memory Pool State

[C Language API]

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

[Parameter]

ID	mplid	ID number of the variable-sized memory pool to be referenced
T_RMPL *	pk_rmpl	Pointer to the packet returning the variable-sized memory pool state

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

pk_rmpl includes (T_RMPL type)

ID	wtskid	ID number of the task at the head of the wait queue
SIZE	fmplsz	Total size of free memory blocks (in bytes)
UINT	fblksz	Maximum memory block size available (in bytes)

(Other implementation specific information may be added.)

[Error Code]

E_ID	Invalid ID number (mplid is invalid or unusable)
E_NOEXS	Non-existent object (specified variable-sized memory pool is not registered)
E_PAR	Parameter error (pk_rmpl is invalid)

[Functional Description]

This service call references the state of the variable-sized memory pool specified by mplid. The state of the memory pool is returned through the packet pointed to by pk_rmpl.

The ID number of the task at the head of the variable-sized memory pool's wait queue is returned through wtskid. If no tasks are waiting to acquire a memory block, TSK_NONE (= 0) is returned instead.

The current total size of free memory blocks in the variable-sized memory pool in bytes is returned through fmplsz.

The size of the largest free memory block in bytes that can be acquired immediately from the variable-sized memory pool is returned through fblksz. When the size of the memory block is too large to represent with UINT type, the maximum value that can fit in UINT type is returned through fblksz.

[Supplemental Information]

If the kernel uses dynamic memory management internally, this service call can be used as an API to reference the kernel's dynamic memory area. Specifically, this service call returns the information on the kernel's dynamic memory area when invoked with an ID number of (–4). However, wtskid does not have a meaning in this case. In addition, if the kernel manages more than one dynamic memory area, these can be referenced through ID numbers (–3) and (–2).

[Differences from the μITRON3.0 Specification]

The extended information has been removed from the reference information. The ID number of the task at the head of the wait queue is now returned instead of a boolean value indicating whether a task is waiting or not. Based on this replacement, the names and data types of the return parameters have been changed.

The names of the return parameters have been changed from frsz to fmplsz and from maxsz to fblksz. The data types of fmplsz and fblksz have been changed from INT to SIZE and from INT to UINT, respectively. The order of parameters and of return parameters has been changed.

4.7 Time Management Functions

Time management functions provide time-dependent processing. The time management functions include system time management, cyclic handlers, alarm handlers, and overrun handlers. Cyclic handlers, alarm handlers, and overrun handlers are generically called time event handlers.

[Supplemental Information]

The contexts and states under which time event handlers execute are summarized as follows:

- Time event handlers execute in their own independent contexts (see Section 3.5.1). The contexts in which time event handlers execute are classified as non-task contexts (see Section 3.5.2).
- Time event handlers execute at lower precedence than the interrupt handler that called isig_tim, but at higher precedence than the dispatcher (see Section 3.5.3).
- After time event handlers start, the system is in the CPU unlocked state. When returning from time event handlers, the system must be in the CPU unlocked state (see Section 3.5.4).
- The start of and the return from time handlers do not change the dispatching state. When the dispatching state is changed within time event handlers, the original state must be restored before returning (see Section 3.5.5).

[Differences from the µITRON3.0 Specification]

The name cyclic handler has been changed from cyclic activation handler. Overrun handler is a newly added feature. The delay task function (del_tsk) has been moved from time management functions to task dependent synchronization functions. ret_tmr has been removed (see Section 3.9).

4.7.1 System Time Management

System time management functions provide control over system time. System time management functions include the ability to set and get the system time and to supply a time tick for updating the system time.

System time initializes to 0 when the system is started (see Section 3.7) and will be updated every time isig_tim is invoked by the application. The amount of time added to the system time when isig_tim is invoked is implementation-defined. The frequency of calling isig_tim from the application must be correlated with the amount of time added to the system time. If the kernel has a mechanism of updating the system time, isig_tim need not be supported.

The following features depend on the system time: processing of timeouts, releasing tasks from waiting after a call to dly_tsk, and activation of cyclic handlers and alarm handlers. The execution order of multiple processes that start at the same system time tick is implementation-dependent.

The following kernel configuration constants are defined for use with system time management functions:

TIC_NUME	Time tick period numerator
TIC_DENO	Time tick period denominator

These constants allow the application to reference the approximate time precision of the system time. TIC_NUME / TIC_DENO is the time tick period measured in the same units as the system time. If the system time is not updated periodically, the constants should still be defined so that they reflect the characteristic of the system time precision.

The following represents the function codes for the system time management service calls:

TFN_SET_TIM	–0x4d	Function code of set_tim
TFN_GET_TIM	–0x4e	Function code of get_tim
TFN_ISIG_TIM	–0x7d	Function code of isig_tim

[Standard Profile]

The Standard Profile requires support for the system time management functions. However, if the kernel has a mechanism of updating the system time, isig_tim need not be supported.

[Supplemental Information]

Another method to define TIC_NUME and TIC_DENO is to allow the application to define them in the system configuration file or in header files prepared by the application. The kernel determines the period that isig_tim is invoked by the application from these constants.

[Differences from the µITRON3.0 Specification]

The name system time has been changed from system clock. The service call to supply a time tick (isig_tim) has been newly added. This allows the kernel to be independent of timer hardware.

The recommended number of bits used to represent the value of the system time is not specified. In the µITRON3.0 Specification it was 48 bits. Now the system is set to 0 upon initialization. In the µITRON3.0 Specification, the recommended start date for absolute time was January 1st, 1985, 0:00 am GMT.

set_tim Set System Time [S]

[C Language API]

ER ercd = set_tim (SYSTIM *p_systim) ;

[Parameter]

SYSTIM	systim	Time to set as system time

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_PAR	Parameter error (p_systim or systim is invalid)

[Functional Description]

This service call sets the system time to the value specified by systim.

[Supplemental Information]

Changing the system time using this service call will not change the time in the real world when an event specified using relative time is to occur. However, the system time when that event occurs will change (see Section 2.1.9).

[Differences from the µITRON3.0 Specification]

The data type of the system time has been changed from SYSTIME to SYSTIM. The parameter name in the C language API has changed from pk_tim to p_systim.

[Rationale]

systim is passed through a pointer because passing the parameter value may reduce system efficiency when SYSTIM is defined as a data structure.

get_tim Reference System Time [S]

[C Language API]

```
ER ercd = get_tim ( SYSTIM *p_systim ) ;
```

[Parameter]

None

[Return Parameter]

ER	ercd	E_OK for normal completion or error code
SYSTIM	systim	Current system time

[Error Code]

E_PAR	Parameter error (p_systim is invalid)

[Functional Description]

This service call returns the current system time through systim.

[Differences from the μITRON3.0 Specification]

The data type of the system time has been changed from SYSTIME to SYSTIM. The parameter name in the C language API has changed from pk_tim to p_systim.

isig_tim Supply Time Tick [S]

[C Language API]

ER ercd = isig_tim ( ) ;

[Parameter]

None

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

No errors specific to this service call

[Functional Description]

This service call updates the system time.

[Standard Profile]

The Standard Profile does not require support for this service call if the kernel has a mechanism of updating the system time.

[Supplemental Information]

This service call may start processes that depend on the system time. This does not mean that these processes must be executed within this service call. This implies that these processes do not necessarily complete before the service call returns.

[Differences from the μITRON3.0 Specification]

This service call has been newly added.

4.7.2 Cyclic Handlers

A cyclic handler is a time event handler activated periodically. Cyclic handler functions include the ability to create and delete a cyclic handler, to start and stop a cyclic handler's operation, and to reference the state of a cyclic handler. A cyclic handler is an object identified by an ID number. The ID number of a cyclic handler is called the cyclic handler ID.

The activation cycle and activation phase are set at the creation of the cyclic handler. The kernel determines the next time the handler will be activated based on the activation cycle and the activation phase. When the cyclic handler is created, the first activation time is calculated by adding the activation phase to the time at which the cyclic handler was created. At the cyclic handler's activation time, the cyclic handler is called with its extended information (exinf) passed as a parameter. At this time the next activation time is calculated by adding the activation cycle to the current activation time. In addition, the next activation time may be recalculated when the cyclic handler's operation is started.

Generally, a cyclic handler's activation phase is less than its activation cycle. The behavior is implementation-dependent when the activation phase is longer than the activation cycle.

A cyclic handler is either in an operational state or a non-operational state. When a cyclic handler is in a non-operational state, the cyclic handler is not activated at its activation time. Instead, its next activation time is determined. When the service call that starts the operation of a cyclic handler (sta_cyc) is invoked, the cyclic handler is moved to an operational state and its next activation time is recalculated if necessary. When the service call that stops the operation of a cyclic handler (stp_cyc) is invoked, the cyclic handler is moved to a non-operational state. After the creation of a cyclic handler, the cyclic handler's attribute determines the operational state of the cyclic handler.

The activation phase is the relative time from the time when the cyclic handler was created to the first activation time. If the cyclic handler is created through a static API, the creation time is considered to be the system initialization time. The activation cycle is the relative time from the last activation time to the next activation time. The last activation time may not have been the actual time of activation, but rather the last expected activation time. An actual interval between actual activations can possibly be shorter than the activation cycle. However, in the long term, the average actual activation interval will correspond with the activation cycle.

The format to write a cyclic handler in the C language is shown below:

```
void cychdr ( VP_INT exinf )
{
        /* Body of the cyclic handler */
}
```

The following data type packets are defined for creating and referencing cyclic handlers:

```
typedef struct t_ccyc {
        ATR         cycatr ;     /* Cyclic handler attribute */
        VP_INT      exinf ;      /* Cyclic handler extended information */
        FP          cychdr ;     /* Cyclic handler start address */
        RELTIM      cyctim ;     /* Cyclic handler activation cycle */
        RELTIM      cycphs ;     /* Cyclic handler activation phase */
        /* Other implementation specific fields may be added. */
} T_CCYC ;

typedef struct t_rcyc {
        STAT        cycstat ;    /* Cyclic handler operational state */
        RELTIM      lefttim ;    /* Time left before the next activation */
        /* Other implementation specific fields may be added. */
} T_RCYC ;
```

The following represents the function codes for the cyclic handler service calls:

TFN_CRE_CYC	–0x4f	Function code of cre_cyc
TFN_ACRE_CYC	–0xcb	Function code of acre_cyc
TFN_DEL_CYC	–0x50	Function code of del_cyc
TFN_STA_CYC	–0x51	Function code of sta_cyc
TFN_STP_CYC	–0x52	Function code of stp_cyc
TFN_REF_CYC	–0x53	Function code of ref_cyc

[Standard Profile]

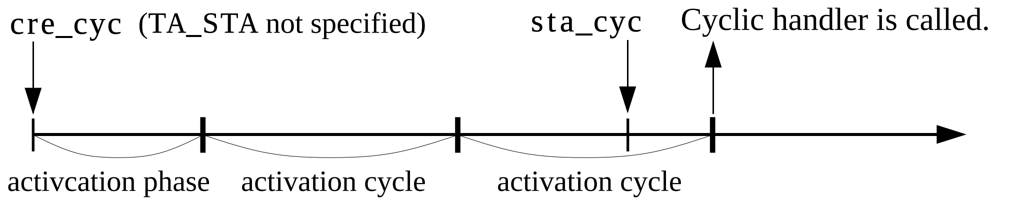
The Standard Profile requires support for cyclic handler functions except for dynamically creation and deletion of a cyclic handler (cre_cyc, acre_cyc, del_cyc) and reference of a cyclic handler state (ref_cyc).

The Standard Profile does not require support for preserving the activation phase, which is specified by TA_PHS in the cyclic handler's attribute.

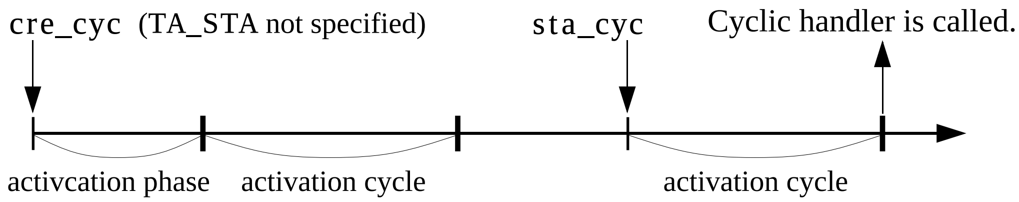
[Supplemental Information]

When the activation phase is preserved, the activation time is determined so that the quantity (((*activation time*) – (*creation time*)) % (*activation cycle*)) is constant. Figure 4-5 show how the cyclic handler is activated after it is created with TA_STP specified in its attribute and then it is moved to an operational state with sta_cyc. When the activation phase is preserved, the activation time is always determined base on the creation time (Figure 4-5 (a)). When the activation phase is not preserved the activation time is determined base on the time when sta_cyc is invoked (Figure 4-5 (b)).

The activation of cyclic handlers depends on the system time. Therefore, these handlers are activated at the first time tick after the activation time has passed. The activation phase is the relative time from when the cyclic handler was created. This means that the first activation of the cyclic handler occurs after an elapsed time equal to or greater than the activation phase (as long as the cyclic handler is in an operational



(a) when the activation phase is preserved (TA_PHS specified)



(b) when the activation phase is not preserved (TA_PHS not specified)

Figure 4-5. Preserving Activation Phase

state). The activation cycle is the relative time from the last activation time. This means that the n-th activation of the cyclic handler must occur after an elapsed time equal to or greater than ((*activation phase*) + (*activation cycle*) * (n–1)) from the creation time. For example, for a system with a 10 millisecond time tick where a cyclic handler is created through the static API with the activation phase set to 15 milliseconds and the activation cycle set to 25 milliseconds, then the activation times will be at 20, 40, 70, 90, and 120 milliseconds and so on. See Section 2.1.9 for how to handle events specified with relative times.

This specification describes the calculation of the next activation time even when a cyclic handler is in a non-operational state. This calculation can be omitted in an implementation as long as the behavior of cyclic handlers do not change.

[Differences from the μITRON3.0 Specification]

The name cyclic handler has been changed from cyclic activation handler. Cyclic handlers are now identified by ID numbers. Cyclic handlers are now objects created by cre_cyc rather than defined by def_cyc. The service call to delete a cyclic handler (del_cyc) has been newly added.

The service call to control the operational state of a cyclic handler (act_cyc) has been divided into a service call that starts the operation of a handler (sta_cyc) and one that stops the operation of a handler (stp_cyc).

CRE_CYC	Create Cyclic Handler (Static API)	[S]
cre_cyc	Create Cyclic Handler	
acre_cyc	Create Cyclic Handler (ID Number Automatic Assignment)	

[Static API]

CRE_CYC (ID cycid, { ATR cycatr, VP_INT exinf, FP cychdr,
RELTIM cyctim, RELTIM cycphs }) ;

[C Language API]

ER ercd = cre_cyc (ID cycid, T_CCYC *pk_ccyc) ;
ER_ID cycid = acre_cyc (T_CCYC *pk_ccyc) ;

[Parameter]

ID	cycid	ID number of the cyclic handler to be created (except acre_cyc)
T_CCYC *	pk_ccyc	Pointer to the packet containing the cyclic handler creation information (In CRE_CYC, the contents must be directly specified.)

pk_ccyc includes (T_CCYC type)

ATR	cycatr	Cyclic handler attribute
VP_INT	exinf	Cyclic handler extended information
FP	cychdr	Cyclic handler start address
RELTIM	cyctim	Cyclic handler activation cycle
RELTIM	cycphs	Cyclic handler activation phase

(Other implementation specific information may be added.)

[Return Parameter]

cre_cyc:

ER	ercd	E_OK for normal completion or error code

acre_cyc:

ER_ID	cycid	ID number (positive value) of the created cyclic handler or error code

[Error Code]

E_ID	Invalid ID number (cycid is invalid or unusable; only cre_cyc)
E_NOID	No ID number available (there is no cyclic handler ID assignable; only acre_cyc)
E_RSATR	Reserved attribute (cycatr is invalid or unusable)
E_PAR	Parameter error (pk_ccyc, cychdr, cyctim, or cycphs is invalid)
E_OBJ	Object state error (cyclic handler is already registered; only

cre_cyc)

[Functional Description]

These service calls create a cyclic handler with an ID number specified by cycid based on the information contained in the packet pointed to by pk_ccyc. cycatr is the attribute of the cyclic handler. exinf is the extended information passed as a parameter to the cyclic handler when it is called. cychdr is the start address of the cyclic handler. cyctim is the activation cycle time. cycphs is the activation phase.

In CRE_CYC, cycid is an integer parameter with automatic assignment. cycatr is a preprocessor constant expression parameter.

acre_cyc assigns a cyclic handler ID from the pool of unassigned cyclic handler IDs and returns the assigned cyclic handler ID.

cycatr can be specified as ((TA_HLNG || TA_ASM) | [TA_STA] | [TA_PHS]). If TA_HLNG (= 0x00) is specified, the cyclic handler is called through the C language interface. If TA_ASM (= 0x01) is specified, the cyclic handler is called through an assembly language interface. If TA_STA (= 0x02) is specified, the handler is in an operational state when it is created, otherwise it is in a non-operational state. If TA_PHS (= 0x04) is specified, the next activation time is determined preserving the activation phase when the cyclic handler is moved to an operational state. See the functional description of sta_cyc for the actions to be taken when a cyclic handler is moved to an operational state.

The first activation time of the cyclic handler is the time when the service call is invoked plus the activation phase. For the static API, the system initialization time is used as the invoking time.

When cyctim is 0, an E_PAR error is returned. The behavior of the system when the value of cycphs is greater than cyctim is implementation-dependent. When an error should be reported, an E_PAR error is returned.

[Standard Profile]

The Standard Profile does not require support for when T_PHS or TA_ASM is specified in cycatr.

[Supplemental Information]

The cyclic handler activation phase (cycphs) does not have any meaning when neither TA_STA nor TA_PHS are specified in cycatr.

[Differences from the μITRON3.0 Specification]

Cyclic handlers are now objects created by cre_cyc rather than defined by def_cyc. The functionality for specifying the activation phase has been newly added. The activation phase (cycphs) has been added to the cyclic handler creation information. The method for specifying the cyclic handler's operational state after creation has been changed.

The order of cycatr and exinf in the creation information packet has been exchanged. The data type of exinf has been changed from VP to VP_INT and the data type of cyctim has been changed from CYCTIME to RELTIM.

acre_cyc has been newly added.

del_cyc Delete Cyclic Handler

[C Language API]

```
ER ercd = del_cyc ( ID cycid ) ;
```

[Parameter]

ID	cycid	ID number of the cyclic handler to be deleted

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (cycid is invalid or unusable)
E_NOEXS	Non-existent object (specified cyclic handler is not registered)

[Functional Description]

This service call deletes the cyclic handler specified by cycid.

[Differences from the μITRON3.0 Specification]

This service call has been newly added. In the μITRON3.0 Specification, the def_cyc service call can be used for releasing a handler as well as defining a handler.

sta_cyc Start Cyclic Handler Operation [S]

[C Language API]

ER ercd = sta_cyc (ID cycid) ;

[Parameter]

ID	cycid	ID number of the cyclic handler operation to be started

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (cycid is invalid or unusable)
E_NOEXS	Non-existent object (specified cyclic handler is not registered)

[Functional Description]

This service call moves the cyclic handler specified by cycid to an operational state.

If the handler's attribute does not have TA_PHS (= 0x04) specified, the next activation time is the time when sta_cyc is invoked plus the activation cycle.

If the cyclic handler is already in an operational state and TA_PHS is not specified in the attribute, the activation time is recalculated. If the cyclic handler is already in an operational state and TA_PHS is specified in the attribute, no action is required.

[Differences from the µITRON3.0 Specification]

The service call to control the operational state of a cyclic handler (act_cyc) has been divided into a service call that starts the operation of a handler (sta_cyc) and one that stops the operation of a handler (stp_cyc). In the µITRON3.0 Specification, when the act_cyc service call is invoked with TCY_INI specified, the activation time is recalculated. A similar functionality is achieved through the use of TA_PHS.

stp_cyc Stop Cyclic Handler Operation [S]

[C Language API]

ER ercd = stp_cyc (ID cycid) ;

[Parameter]

ID	cycid	ID number of the cyclic handler operation to be stopped

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (cycid is invalid or unusable)
E_NOEXS	Non-existent object (specified cyclic handler is not registered)

[Functional Description]

This service call moves the cyclic handler specified by cycid to a non-operational state. No action is required when the specified cyclic handler is already in a non-operational state.

[Differences from the μITRON3.0 Specification]

The service call to control the operational state of a cyclic handler (act_cyc) has been divided into a service call that starts the operation of a handler (sta_cyc) and one that stops the operation of a handler (stp_cyc).

ref_cyc Reference Cyclic Handler State

[C Language API]

```
ER ercd = ref_cyc ( ID cycid, T_RCYC *pk_rcyc ) ;
```

[Parameter]

ID	cycid	ID number of the cyclic handler to be referenced
T_RCYC *	pk_rcyc	Pointer to the packet returning the cyclic handler state

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

pk_rcyc includes (T_RCYC type)

STAT	cycstat	Cyclic handler operational state
RELTIM	lefttim	Time left before the next activation

(Other implementation specific information may be added.)

[Error Code]

E_ID	Invalid ID number (cycid is invalid or unusable)
E_NOEXS	Non-existent object (specified cyclic handler is not registered)
E_PAR	Parameter error (pk_rcyc is invalid)

[Functional Description]

This service call references the state of the cyclic handler specified by cycid. The state of the cyclic handler is returned through the packet pointed to by pk_rcyc.

One of the following values is returned through cycstat depending on the operational state of the cyclic handler:

TCYC_STP	0x00	Cyclic handler is in a non-operational state
TCYC_STA	0x01	Cyclic handler is in an operational state

The amount of time remaining before the cyclic handler's next activation time is returned through lefttim if the cyclic handler is in an operational state. This means the time returned is the next activation time minus the current time. The value returned will be less than the time it will take to activate the cyclic handler. Therefore, if 0 is returned, the cyclic handler will be activated on the next time tick. The value returned through lefttim when the cyclic handler is a non-operational state is implementation-dependent.

[Differences from the μITRON3.0 Specification]

The extended information has been removed from the reference information. The method to reference the operational state has been changed. The data type of lefttim has been changed from CYCTIME to RELTIM. The order of parameters and of return parameters has been changed.

4.7.3 Alarm Handlers

An alarm handler is a time event handler activated at a specified time. Alarm handler functions include the ability to create and delete an alarm handler, to start and stop an alarm handler's operation, and to reference the state of an alarm handler. An alarm handler is an object identified by an ID number. The ID number of an alarm handler is called the alarm handler ID.

The time at which the alarm handler is activated, called the activation time of the alarm handler, can be set for each handler. At the alarm handler's activation time, the alarm handler is called with its extended information (exinf) passed as a parameter.

The activation time of the alarm handler is not set when the handler is created. Therefore, the operation of the alarm handler is stopped. The service call that starts the operation of an alarm handler (sta_alm) sets the activation time relative to the time when the service call is invoked. In addition, the alarm handler is moved to an operational state. When the service call that stops the operation of an alarm handler (stp_alm) is invoked, the activation time is released and the alarm handler is moved to a non-operational state. When an alarm handler is called, the activation time is released and the alarm handler is moved to a non-operational state.

The format to write an alarm handler in the C language is shown below:

```
void almhdr ( VP_INT exinf )
{
        /* Body of the alarm handler */
}
```

The following data type packets are defined for creating and referencing alarm handlers:

```
typedef struct t_calm {
        ATR         almatr ;    /* Alarm handler attribute */
        VP_INT      exinf ;     /* Alarm handler extended information */
        FP          almhdr ;    /* Alarm handler start address */
        /* Other implementation specific fields may be added. */
} T_CALM ;

typedef struct t_ralm {
        STAT        almstat ;   /* Alarm handler operational state */
        RELTIM      lefttim ;   /* Time left before the activation */
        /* Other implementation specific fields may be added. */
} T_RALM ;
```

The following represents the function codes for the alarm handler service calls:

TFN_CRE_ALM	–0xa9	Function code of cre_alm
TFN_ACRE_ALM	–0xcc	Function code of acre_alm
TFN_DEL_ALM	–0xaa	Function code of del_alm
TFN_STA_ALM	–0xab	Function code of sta_alm
TFN_STP_ALM	–0xac	Function code of stp_alm

TFN_REF_ALM	–0xad	Function code of ref_alm

[Standard Profile]

The Standard Profile does not require support for alarm handlers.

[Supplemental Information]

The activation of alarm handlers depends on the system time. Therefore, these handlers are activated at the first time tick after the activation time has passed. The system must guarantee that the activation of the alarm handler occurs after an elapsed time equal to or greater than the specified time (see Section 2.1.9).

The activation time is released when the alarm handler is called but before the alarm handler is executed. If an implementation allows non-task contexts to invoke the service call to start the alarm handler operation, the alarm handler can reset the activation time and move itself to an operational state.

[Differences from the μITRON3.0 Specification]

Alarm handlers are now identified by ID numbers. Alarm handlers are now objects created by cre_alm rather than defined by def_alm. The service call to delete an alarm handler (del_alm) has been newly added.

For the case when an alarm handler is created statically, the activation time of the alarm handler is now specified with the newly added service call (sta_alm) instead of the create alarm handler service call or the static API. The service call to stop the operation of a alarm handler (stp_alm) has been newly added.

The ability to set an alarm handler activation time to an absolute time has been removed.

CRE_ALM	Create Alarm Handler (Static API)
cre_alm	Create Alarm Handler
acre_alm	Create Alarm Handler (ID Number Automatic Assignment)

[Static API]

CRE_ALM (ID almid, { ATR almatr, VP_INT exinf, FP almhdr }) ;

[C Language API]

ER ercd = cre_alm (ID almid, T_CALM *pk_calm) ;

ER_ID almid = acre_alm (T_CALM *pk_calm) ;

[Parameter]

ID	almid	ID number of the alarm handler to be created (except acre_alm)
T_CALM *	pk_calm	Pointer to the packet containing the alarm handler creation information (In CRE_ALM, the contents must be directly specified.)

pk_calm includes (T_CALM type)

ATR	almatr	Alarm handler attribute
VP_INT	exinf	Alarm handler extended information
FP	almhdr	Alarm handler start address

(Other implementation specific information may be added.)

[Return Parameter]

cre_alm:

ER	ercd	E_OK for normal completion or error code

acre_alm:

ER_ID	almid	ID number (positive value) of the created alarm handler or error code

[Error Code]

E_ID	Invalid ID number (almid is invalid or unusable; only cre_alm)
E_NOID	No ID number available (there is no alarm handler ID assignable; only acre_alm)
E_RSATR	Reserved attribute (almatr is invalid or unusable)
E_PAR	Parameter error (pk_calm or almhdr is invalid)
E_OBJ	Object state error (alarm handler is already registered; only cre_alm)

[Functional Description]

These service calls create an alarm handler with an ID number specified by almid based on the information contained in the packet pointed to by pk_calm. almatr is the attribute of the alarm handler. exinf is the extended information passed as a parameter to the alarm handler when it is called. almhdr is the start address of the alarm handler.

In CRE_ALM, almid is an integer parameter with automatic assignment. almatr is a preprocessor constant expression parameter.

acre_alm assigns an alarm handler ID from the pool of unassigned alarm handler IDs and returns the assigned alarm handler ID.

After the alarm handler is created, the activation time is not set and the alarm handler is in a non-operational state.

almatr can be specified as (TA_HLNG || TA_ASM). If TA_HLNG (= 0x00) is specified, the alarm handler is called through the C language interface. If TA_ASM (= 0x01) is specified, the alarm handler is called through an assembly language interface.

[Differences from the µITRON3.0 Specification]

Alarm handlers are now objects created by cre_alm rather than defined by def_alm. For the case when an alarm handler is created statically, the activation time of the alarm handler is not specified by the create alarm handler service call or the static API.

The order of almatr and exinf in the creation information packet has been exchanged. The data type of exinf has been changed from VP to VP_INT.

acre_alm has been newly added.

del_alm Delete Alarm Handler

[C Language API]

ER ercd = del_alm (ID almid) ;

[Parameter]

ID	almid	ID number of the alarm handler to be deleted

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (almid is invalid or unusable)
E_NOEXS	Non-existent object (specified alarm handler is not registered)

[Functional Description]

This service call deletes the alarm handler specified by almid.

[Supplemental Information]

If the alarm handler is in an operational state, the activation time is released and the alarm handler is moved to a non-operational state.

[Differences from the μITRON3.0 Specification]

This service call has been newly added. In the μITRON3.0 Specification, the def_alm service call can be used for releasing a handler as well as defining a handler.

sta_alm Start Alarm Handler Operation

[C Language API]

ER ercd = sta_alm (ID almid, RELTIM almtim) ;

[Parameter]

ID	almid	ID number of the alarm handler operation to be started
RELTIM	almtim	Activation time of the alarm handler (relative time)

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (almid is invalid or unusable)
E_NOEXS	Non-existent object (specified alarm handler is not registered)
E_PAR	Parameter error (almtim is invalid)

[Functional Description]

This service call sets the activation time of the alarm handler specified by almid. The activation time is set to the time when the service call is invoked plus the relative time specified by almtim. The alarm handler is also moved to an operational state.

If the alarm handler is already in an operational state, the previous activation time is released and a new activation time is set.

almtim is the relative time from when this service call is invoked to the activation time of the alarm handler.

[Differences from the µITRON3.0 Specification]

This service call has been newly added. The µITRON3.0 Specification allowed def_alm to set the activation time of an alarm handler.

stp_alm Stop Alarm Handler Operation

[C Language API]

```
ER ercd = stp_alm ( ID almid ) ;
```

[Parameter]

ID	almid	ID number of the alarm handler operation to be stopped

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (almid is invalid or unusable)
E_NOEXS	Non-existent object (specified alarm handler is not registered)

[Functional Description]

This service call releases the activation time of the alarm handler specified by almid and moves the alarm handler to a non-operational state. If the alarm handler is already in a non-operational state, no action is required.

[Differences from the µITRON3.0 Specification]

This service call has been newly added. The µITRON3.0 specification did not allow an alarm handler to be stopped by any other means than releasing the registration of the alarm handler.

ref_alm Reference Alarm Handler State

[C Language API]

```
ER ercd = ref_alm ( ID almid, T_RALM *pk_ralm ) ;
```

[Parameter]

ID	almid	ID number of the alarm handler to be referenced
T_RALM *	pk_ralm	Pointer to the packet returning the alarm handler state

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

pk_ralm includes (T_RALM type)

STAT	almstat	Alarm handler operational state
RELTIM	lefttim	Time left before the activation

(Other implementation specific information may be added.)

[Error Code]

E_ID	Invalid ID number (almid is invalid or unusable)
E_NOEXS	Non-existent object (specified alarm handler is not registered)
E_PAR	Parameter error (pk_ralm is invalid)

[Functional Description]

This service call references the state of the alarm handler specified by almid. The state of the alarm handler is returned through the packet pointed to by pk_ralm.

One of the following values will be returned through almstat depending on the operational state of the alarm handler:

TALM_STP	0x00	Alarm handler is in a non-operational state
TALM_STA	0x01	Alarm handler is in an operational state

The amount of time remaining before the alarm handler's activation time is returned through lefttim if the alarm handler is in an operational state. This means the time returned is the activation time minus the current time. The value returned will be less than the time it will take to activate the alarm handler. Therefore, if 0 is returned, the alarm handler will be activated on the next time tick. The value returned through lefttim when the alarm handler is a non-operational state is implementation-dependent.

[Differences from the µITRON3.0 Specification]

The alarm handler operational state (almstat) has been added to the reference information. The extended information has been removed from the reference information. The data type of lefttim has been changed from ALMTIME to RELTIM. The order of the parameters and of the return parameters has been changed.

4.7.4 Overrun Handler

The overrun handler is a time event handler activated when a task has been executed by the processor longer than a specified amount of time. Overrun handler functions include the ability to define the overrun handler, to start and stop the overrun handler's operation, and to reference the state of the overrun handler.

The amount of time used to determine the activation condition, called the processor time limit, can be specified for each task. Once a task has a processor time limit set, the kernel keeps track of the accumulated processor time consumed by the task, called the processor time used, until the consumed time exceeds the time limit. Once this occurs, the overrun handler is called. Because only one overrun handler can be defined for the whole system, the task ID number (tskid) and the task's extended information (exinf) are passed as parameters to the overrun handler.

The task's processor time limit is not set when the task is created. When the service call to start the overrun handler operation (sta_ovr) is invoked for a specified task, the processor time limit is set for the task. In addition, the processor time used for the task is cleared to 0. Once the service call to stop the overrun handler operation (stp_ovr) is invoked for a specified task, the processor time limit for the task is released. The processor time limit for a task is also released when the overrun handler is called for the task or when the task is terminated.

The processor time used by a task includes the time consumed by the task, by the task's exception handling routine, and by all service calls invoked by the task. On the other hand, the time consumed by the other tasks, by their exception handling routines, and by all the service calls they invoke are not included in the processor time used by the task. The decision to include the time for task dispatching and for interrupt processing is implementation-dependent. In addition, the accuracy of the measured processor time used is implementation-dependent. Nevertheless, the overrun handler is activated only when the processor time used exceeds the specified processor time limit.

The following data type is used within the overrun handler functions:

OVRTIM	Processor time (unsigned integer, unit of time is implementation-defined)

The format to write an overrun handler in the C language is shown below:

```
void ovrhdr ( ID tskid, VP_INT exinf )
{
        /* Body of the overrun handler */
}
```

The following data type packets are defined for defining and referencing overrun handlers:

```
typedef struct t_dovr {
        ATR     ovratr ;    /* Overrun handler attribute */
        FP      ovrhdr ;    /* Overrun handler start address */
```

```
        /* Other implementation specific fields may be added. */
} T_DOVR ;

typedef struct t_rovr {
        STAT        ovrstat ;     /* Overrun handler operational state */
        OVRTIM      leftotm ;     /* Remaining processor time */
        /* Other implementation specific fields may be added. */
} T_ROVR ;
```

The following represents the function codes for the overrun handler service calls:

TFN_DEF_OVR	–0xb1	Function code of def_ovr
TFN_STA_OVR	–0xb2	Function code of sta_ovr
TFN_STP_OVR	–0xb3	Function code of stp_ovr
TFN_REF_OVR	–0xb4	Function code of ref_ovr

[Standard Profile]

The Standard Profile does not require support for the overrun handler.

[Supplemental Information]

The activation of the overrun handler does not depend on the system time. This implies the handler is not necessarily called synchronously with the time tick. Implementations may call the overrun handler synchronously with the time tick.

A task's processor time limit is released when the handler is called but before the overrun handler is executed. If an implementation allows non-task contexts to invoke the service call to start the overrun handler operation, the overrun handler can reset the processor time limit for the task that causes the overrun handler's activation.

The overrun handler can raise a task's exception. Then, the task's exception handling routine is started by the kernel within the task's context to handle the overrun situation.

[Differences from the μITRON3.0 Specification]

Overrun handler is a newly added feature.

DEF_OVR Define Overrun Handler (Static API)
def_ovr Define Overrun Handler

[Static API]

DEF_OVR ({ ATR ovratr, FP ovrhdr }) ;

[C Language API]

ER ercd = def_ovr (T_DOVR *pk_dovr) ;

[Parameter]

T_DOVR *	pk_dovr	Pointer to the packet containing the overrun handler definition information (in DEF_OVR, the contents must be directly specified.)

pk_dovr includes (T_DOVR type)

ATR	ovratr	Overrun handler attribute
FP	ovrhdr	Overrun handler start address

(Other implementation specific information may be added.)

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_RSATR	Reserved attribute (ovratr is invalid or unusable)
E_PAR	Parameter error (pk_dovr or ovrhdr is invalid)

[Functional Description]

This service call defines the overrun handler based on the information contained in the packet pointed to by pk_dovr. ovratr is the attribute of the overrun handler. ovrhdr is the start address of the overrun handler.

In DEF_OVR, ovratr is a preprocessor constant expression parameter.

If pk_dovr is NULL (= 0), the overrun handler currently defined is released and the overrun handler becomes undefined. At this time, the processor time limits for all tasks are also released. When a new overrun handler is defined over top of an old one, the old one is released and the new one takes its place. Under this condition, the processor time limits for the tasks are not released.

ovratr can be specified as (TA_HLNG || TA_ASM). If TA_HLNG (= 0x00) is specified, the overrun handler is called through the C language interface. If TA_ASM (= 0x01) is specified, the overrun handler is called through an assembly language interface.

[Rationale]

The reason why the processor time limit is released for a task when the definition of the handler is released is to ensure that there is no processor time limit set while the overrun handler is undefined.

sta_ovr Start Overrun Handler Operation

[C Language API]

ER ercd = sta_ovr (ID tskid, OVRTIM ovrtim) ;

[Parameter]

ID	tskid	ID number of the task where the overrun handler should start operation
OVRTIM	ovrtim	Processor time limit for the task to be set

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (tskid is invalid or unusable)
E_NOEXS	Non-existent object (specified task is not registered)
E_PAR	Parameter error (ovrtim is invalid)
E_OBJ	Object state error (overrun handler is not defined)

[Functional Description]

This service call starts the operation of the overrun handler for the task specified by tskid. It also sets the processor time limit for the task as specified by ovrtim. In addition, the processor time used by the task is cleared to 0.

Even if the task already has a processor time limit set, the processor time limit will be reset to the new value and the processor time used will be cleared to 0.

If tskid is TSK_SELF (= 0), the task that invoked the service call will be the target task.

stp_ovr Stop Overrun Handler Operation

[C Language API]

ER ercd = stp_ovr (ID tskid) ;

[Parameter]

ID	tskid	ID number of the task on which the overrun handler should stop operation

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (tskid is invalid or unusable)
E_NOEXS	Non-existent object (specified task is not registered)
E_OBJ	Object state error (overrun handler is not defined)

[Functional Description]

This service call stops the operation of the overrun handler for the task specified by tskid by releasing the processor time limit for the task. If the specified task does not have a processor time limit set, no action is required.

If tskid is TSK_SELF (= 0), the task that invoked the service call will be the target task.

ref_ovr Reference Overrun Handler State

[C Language API]

ER ercd = ref_ovr (ID tskid, T_ROVR *pk_rovr) ;

[Parameter]

ID	tskid	ID number of the task for which the overrun handler's state should be referenced
T_ROVR *	pk_rovr	Pointer to the packet returning the overrun handler state

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

pk_rovr includes (T_ROVR type)

STAT	ovrstat	Overrun handler operational state
OVRTIM	leftotm	Remaining processor time

(Other implementation specific information may be added.)

[Error Code]

E_ID	Invalid ID number (tskid is invalid or unusable)
E_NOEXS	Non-existent object (specified task is not registered)
E_PAR	Parameter error (pk_rovr is invalid)
E_OBJ	Object state error (overrun handler is not defined)

[Functional Description]

This service call references the state of the overrun handler for the task specified by tskid. The state of the overrun handler is returned through the packet pointed to by pk_rovr.

The operational state of the overrun handler is returned through ovrstat. One of the following values is returned depending on whether the processor time limit has been set for the task:

TOVR_STP	0x00	Processor time limit is not set
TOVR_STA	0x01	Processor time limit is set

The processor time remaining until the overrun handler is called for the specified task is returned through leftotm if the processor time limit is set for the specified task. This means the value returned is the processor time limit minus the processor time used. The value returned will be less than the actual remaining processor time which can be consumed by the task until the overrun handler is called. Therefore, 0 can be returned through leftotm if this service call is invoked just before the overrun handler is called. The value returned through leftotm when the processor time limit is not set for the specified task is implementation-dependent.

If tskid is TSK_SELF (= 0), the task that invoked the service call will be the target task.

4.8 System State Management Functions

System state management functions provide control of and reference to the various system states. System state management functions include the ability to rotate task precedence, to reference the ID of the task in the RUNNING state, to lock and unlock the CPU, to enable and disable dispatching, and to reference the context and the system state.

The following data type packet is defined for referencing system state:

```
typedef struct t_rsys {
        /* Implementation specific fields */
} T_RSYS ;
```

The following represents the function codes for the system state management service calls:

TFN_ROT_RDQ	–0x55	Function code of rot_rdq
TFN_IROT_RDQ	–0x79	Function code of irot_rdq
TFN_GET_TID	–0x56	Function code of get_tid
TFN_IGET_TID	–0x7a	Function code of iget_tid
TFN_LOC_CPU	–0x59	Function code of loc_cpu
TFN_ILOC_CPU	–0x7b	Function code of iloc_cpu
TFN_UNL_CPU	–0x5a	Function code of unl_cpu
TFN_IUNL_CPU	–0x7c	Function code of iunl_cpu
TFN_DIS_DSP	–0x5b	Function code of dis_dsp
TFN_ENA_DSP	–0x5c	Function code of ena_dsp
TFN_SNS_CTX	–0x5d	Function code of sns_ctx
TFN_SNS_LOC	–0x5e	Function code of sns_loc
TFN_SNS_DSP	–0x5f	Function code of sns_dsp
TFN_SNS_DPN	–0x60	Function code of sns_dpn
TFN_REF_SYS	–0x61	Function code of ref_sys

[Standard Profile]

The Standard Profile requires support for system state management functions except for the reference of the system state (ref_sys).

[Differences from the μITRON3.0 Specification]

The category of system state management functions has been newly added.

rot_rdq Rotate Task Precedence [S]
irot_rdq [S]

[C Language API]

```
ER ercd = rot_rdq ( PRI tskpri ) ;
ER ercd = irot_rdq ( PRI tskpri ) ;
```

[Parameter]

PRI	tskpri	Priority of the tasks whose precedence is rotated

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_PAR	Parameter error (tskpri is invalid)

[Functional Description]

These service calls rotate the precedence of the tasks with the priority specified by tskpri. In other words, the task with the highest precedence of all the runnable tasks with the specified priority will have the lowest precedence among the tasks with the same priority after the precedence rotation.

If tskpri is TPRI_SELF (= 0), the base priority of the invoking task becomes the target priority. An E_PAR error is returned if TPRI_SELF is specified when the service call is invoked from non-task contexts.

[Supplemental Information]

Round-robin scheduling can be achieved by invoking this service call periodically. No action is required if there is a single task at the target priority or no tasks at the target priority (no error is reported).

When the service call is invoked with the current priority of the invoking task as the target priority while in the dispatching enabled state, the invoking task's precedence becomes the lowest among the tasks with the same priority. This means the invoking task may yield it execution privilege to another task. While in the dispatching disabled state, the task with the highest precedence among the tasks with the same priority may not necessarily be the running task. Therefore, the invoking task's precedence may not become the lowest among the tasks with the same priority using this yield method. The yield method can be realized by invoking the service call with TPRI_SELF specified for tskpri when the current priority of the invoking task equals its base priority, as is always the case when mutex functions are not used.

[Differences from the µITRON3.0 Specification]

The ability to rotate the tasks precedence at the running task's priority from non-task contexts has been removed. Therefore, TPRI_RUN has been changed to

TPRI_SELF. TPRI_SELF now specifies the base priority of the invoking task due to the introduction of mutex functions.

get_tid Reference Task ID in the RUNNING State [S]
iget_tid [S]

[C Language API]

```
ER ercd = get_tid ( ID *p_tskid ) ;
ER ercd = iget_tid ( ID *p_tskid ) ;
```

[Parameter]

None

[Return Parameter]

ER	ercd	E_OK for normal completion or error code
ID	tskid	ID number of the task in the RUNNING state

[Error Code]

No errors specific to this service call

[Functional Description]

These service calls reference the ID number of the task in the RUNNING state (this corresponds to the invoking task when the service call is invoked from task contexts) and return the task ID through tskid. If no task is in the RUNNING state when the service call is invoked from non-task contexts, TSK_NONE (= 0) is returned instead.

[Supplemental Information]

Some kernel implementations employ an idle task that runs when no application tasks are runnable. When the service call is invoked for such a kernel implementation while an idle task is in the RUNNING state, TSK_NONE is returned instead of the ID number of the idle task.

[Differences from the μITRON3.0 Specification]

This service call has been changed from returning the invoking task ID to returning the task ID of the task in the RUNNING state. As a result, the behavior upon invoking this service call from non-task contexts has been changed.

[Rationale]

The reason why tskid is not returned through the return value of the service call is because negative task ID numbers can be supported.

loc_cpu Lock the CPU [S]
iloc_cpu [S]

[C Language API]

```
ER ercd = loc_cpu ( ) ;
ER ercd = iloc_cpu ( ) ;
```

[Parameter]

None

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

No errors specific to this service call

[Functional Description]

These service calls transition the system to the CPU locked state. If the system is in the CPU locked state, no action is required.

[Supplemental Information]

The system is released from the CPU locked state when unl_cpu or iunl_cpu is invoked once, even if multiple calls of loc_cpu or iloc_cpu have been made. Therefore, if a pair of loc_cpu or iloc_cpu and unl_cpu or iunl_cpu need to be nested, the following method may be required:

```
{
        BOOL cpu_locked = sns_loc ( ) ;

        if ( !cpu_locked )
                loc_cpu ( ) ;
        /* work to do in the CPU locked state */
        if ( !cpu_locked )
                unl_cpu ( ) ;
}
```

[Differences from the μITRON3.0 Specification]

The meaning of the CPU locked state has been changed (see Section 3.5.4). In addition, the service call may now be invoked from non-task contexts.

unl_cpu Unlock the CPU [S]
iunl_cpu [S]

[C Language API]

```
ER ercd = unl_cpu ( ) ;
ER ercd = iunl_cpu ( ) ;
```

[Parameter]

None

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

No errors specific to this service call

[Functional Description]

These service calls transition the system to the CPU unlocked state. If the system is in the CPU unlocked state, no action is required.

[Differences from the µITRON3.0 Specification]

The meaning of the CPU unlocked state has been changed (see Section 3.5.4). Now, invoking this service call does not necessarily transition the system to the dispatching enabled state. In addition, the service call may now be invoked from non-task contexts.

dis_dsp Disable Dispatching [S]

[C Language API]

```
ER ercd = dis_dsp ( ) ;
```

[Parameter]

None

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

No errors specific to this service call

[Functional Description]

This service call transitions the system to the dispatching disabled state. If the system is in the dispatching disabled state, no action is required.

[Supplemental Information]

The system is released from the dispatching disabled state when ena_dsp is invoked once, even if multiple calls of dis_dsp have been made. Therefore, if a pair of dis_dsp and ena_dsp need to be nested, the following method may be required:

```
{
        BOOL dispatch_disabled = sns_dsp ( ) ;

        if ( !dispatch_disabled )
                dis_dsp ( ) ;
        /* work to do in the dispatching disabled state */
        if ( !dispatch_disabled )
                ena_dsp ( ) ;
}
```

[Differences from the µITRON3.0 Specification]

The meaning of the dispatching state has been changed (see Section 3.5.5).

ena_dsp Enable Dispatching [S]

[C Language API]

```
ER ercd = ena_dsp ( ) ;
```

[Parameter]

None

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

No errors specific to this service call

[Functional Description]

This service call transitions the system to the dispatching enabled state. If the system is in the dispatching enabled state, no action is required.

[Differences from the μITRON3.0 Specification]

The meaning of the dispatching state has been changed (see Section 3.5.5).

sns_ctx Reference Contexts [S]

[C Language API]

```
BOOL state = sns_ctx ( ) ;
```

[Parameter]

None

[Return Parameter]

BOOL	state	Context

[Functional Description]

This service call returns TRUE if invoked from non-task contexts and returns FALSE if invoked from task contexts.

[Differences from the μITRON3.0 Specification]

This service call has been newly added.

sns_loc Reference CPU State [S]

[C Language API]

BOOL state = sns_loc ( ) ;

[Parameter]

None

[Return Parameter]

BOOL	state	CPU state

[Functional Description]

This service call returns TRUE if the system is in the CPU locked state and returns FALSE if the system is in the CPU unlocked state.

[Differences from the µITRON3.0 Specification]

This service call has been newly added.

sns_dsp Reference Dispatching State [S]

[C Language API]

```
BOOL state = sns_dsp ( ) ;
```

[Parameter]

None

[Return Parameter]

BOOL	state	Dispatching state

[Functional Description]

This service call returns TRUE if the system is in the dispatching disabled state and returns FALSE if the system is in the dispatching enabled state.

[Differences from the µITRON3.0 Specification]

This service call has been newly added.

sns_dpn Reference Dispatch Pending State [S]

[C Language API]

BOOL state = sns_dpn ( ) ;

[Parameter]

None

[Return Parameter]

BOOL	state	Dispatch pending state

[Functional Description]

This service call returns TRUE if the system is in the dispatch pending state and returns FALSE in any other states. In other words, it returns TRUE, while a processing unit with higher precedence than the dispatcher is executing, while in the CPU locked state, or while in the dispatching disabled state.

[Supplemental Information]

If the system is in the condition where this service call returns FALSE, those service calls which possibly put the invoking task into the WAITING state may be invoked.

[Differences from the µITRON3.0 Specification]

This service call has been newly added.

ref_sys Reference System State

[C Language API]

ER ercd = ref_sys (T_RSYS *pk_rsys) ;

[Parameter]

T_RSYS *	pk_rsys	Pointer to the packet returning the system state

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

pk_rsys includes (T_RSYS type)

(Implementation-specific information)

[Error Code]

E_PAR	Parameter error (pk_rsys is invalid)

[Functional Description]

This service call references the system state and returns it through the packet pointed to by pk_rsys. The specific information referenced is implementation-defined.

[Supplemental Information]

Possible information that may be referenced by this service call includes: states which can be referenced by other reference service calls (get_tid, sns_ctx, sns_loc, sns_dsp, sns_dpn), priority of the task in the RUNNING state, interrupt enabled or disabled state, interrupt mask, processor execution mode, and other information depending on the target processor's architecture.

[Differences from the µITRON3.0 Specification]

In the µITRON4.0 Specification, the information returned by the reference service calls (sns_ctx, sns_loc, sns_dsp) replace the information returned by ref_sys (sysstat) in the µITRON3.0 Specification.

4.9 Interrupt Management Functions

Interrupt management functions provide management for interrupt handlers and for interrupt service routines started by external interrupts. The interrupt management functions include ability to define an interrupt handler, to create and delete an interrupt service routine, to reference the state of an interrupt service routine, to disable and enable an interrupt, and to change and reference the interrupt mask. An interrupt service routine is an object identified by an ID number. The ID number of an interrupt service routine is called the interrupt service routine ID.

The following data types are used for interrupt management functions:

INHNO	Interrupt handler number
INTNO	Interrupt number
I*XXXX*	Interrupt mask

The *XXXX* portion of the interrupt mask data type is implementation-defined and should be an appropriate character string for the target processor's architecture.

The format to write an interrupt handler is implementation-defined.

When calling an interrupt service routine, the extended information (exinf) of the interrupt service routine is passed as a parameter. The format to write an interrupt service routine in the C language is shown below:

```
void isr ( VP_INT exinf )
{
        /* Body of the interrupt service routine */
}
```

The following data type packets are defined for defining interrupt handlers and for creating and referencing interrupt service routines:

```
typedef struct t_dinh {
        ATR        inhatr ;     /* Interrupt handler attribute */
        FP         inthdr ;     /* Interrupt handler start address */
        /* Other implementation specific fields may be added. */
} T_DINH ;

typedef struct t_cisr {
        ATR        isratr ;     /* Interrupt service routine attribute */
        VP_INT     exinf ;      /* Interrupt service routine extended
                                   information */
        INTNO      intno ;      /* Interrupt number to which the interrupt
                                   service routine is to be attached */
        FP         isr ;        /* Interrupt service routine start sddress */
        /* Other implementation specific fields may be added. */
} T_CISR ;

typedef struct t_risr {
        /* Implementation-specific fields */
} T_RISR ;
```

The following represents the function codes for the interrupt management service calls:

TFN_DEF_INH	–0x65	Function code of def_inh
TFN_CRE_ISR	–0x66	Function code of cre_isr
TFN_ACRE_ISR	–0xcd	Function code of acre_isr
TFN_DEL_ISR	–0x67	Function code of del_isr
TFN_REF_ISR	–0x68	Function code of ref_isr
TFN_DIS_INT	–0x69	Function code of dis_int
TFN_ENA_INT	–0x6a	Function code of ena_int
TFN_CHG_IXX	–0x6b	Function code of chg_ixx
TFN_GET_IXX	–0x6c	Function code of get_ixx

[Standard Profile]

The Standard Profile requires support for the static API to define an interrupt handler (DEF_INH). If the implementation supports the static API that attaches an interrupt service routine to the kernel (ATT_ISR), the implementation does not have to support DEF_INH.

[Supplemental Information]

The contexts and states under which interrupt handlers execute are summarized as follows:

- Interrupt handlers execute in their own independent contexts (see Section 3.5.1). The contexts in which interrupt handlers execute are classified as non-task contexts (see Section 3.5.2).
- Interrupt handlers execute at higher precedence than the dispatcher (see Section 3.5.3).
- After interrupt handlers start, whether the system is in the CPU locked state or in the CPU unlocked state is implementation-dependent. However, the implementation must provide a means to unlock the CPU in an interrupt service routine as well as a means to correctly return from the interrupt handler after unlocking the CPU (see Section 3.5.4).
- The start of and the return from interrupt handlers do not change the dispatching state. When the dispatching state is changed within interrupt handlers, the original state must be restored before returning (see Section 3.5.5).

The contexts and states under which interrupt service routines execute are summarized as follows:

- Interrupt service routines execute in their own independent contexts (see Section 3.5.1). The contexts in which interrupt service routines execute are classified as non-task contexts (see Section 3.5.2).
- Interrupt service routines execute at higher precedence than the dispatcher (see Section 3.5.3).
- After interrupt service routines start, the system is in the CPU unlocked state. When

returning from interrupt service routines, the system must be in the CPU unlocked state (see Section 3.5.4).

- The start of and the return from interrupt service routines do not change the dispatching state. When the dispatching state is changed within interrupt service routines, the original state must be restored before returning (see Section 3.5.5).

[Differences from the μITRON3.0 Specification]

loc_cpu and unl_cpu are now classified as system state management functions. ret_int and ret_wup have been removed (see Section 3.9).

The data type of the parameter and the return parameter for an interrupt mask has been changed from UINT to a newly added data type IXXXX.

DEF_INH Define Interrupt Handler (Static API) [S]
def_inh Define Interrupt Handler

[Static API]

DEF_INH (INHNO inhno, { ATR inhatr, FP inthdr }) ;

[C Language API]

ER ercd = def_inh (INHNO inhno, T_DINH *pk_dinh) ;

[Parameter]

INHNO	inhno	Interrupt handler number to be defined
T_DINH *	pk_dinh	Pointer to the packet containing the interrupt handler definition information (in DEF_INH, packet contents must be directly specified.)

pk_dinh includes (T_DINH type)

ATR	inhatr	Interrupt handler attribute
FP	inthdr	Interrupt handler start address

(Other implementation specific information may be added.)

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_RSATR	Reserved attribute (inhatr is invalid or unusable)
E_PAR	Parameter error (inhno, pk_dinh, or inthdr is invalid)

[Functional Description]

This service call assigns an interrupt handler to the interrupt handler number specified by inhno based on the information contained in the packet pointed to by pk_dinh. inhatr is the interrupt handler attribute. inthdr is the start address of the interrupt handler.

In DEF_INH, inhno is an integer parameter without automatic assignment. inhatr is a preprocessor constant expression parameter.

The specific meaning of inhno is implementation-defined, but it corresponds to the processor's interrupt vector number in typical implementations. If a processor does not have interrupt vectors, only one interrupt handler number may be available.

If pk_dinh is NULL (= 0), the interrupt handler currently defined is released. When a new interrupt handler is defined over top of an old one, the old one is released and the new takes its place.

The possible values and meanings of inhatr are implementation-defined.

[Standard Profile]

The Standard Profile does not require support for DEF_INH if the implementation supports ATT_ISR.

[Differences from the µITRON3.0 Specification]

The abbreviation of interrupt handler has been changed from int to inh. Therefore, the name of this service call has been changed from def_int to def_inh. The possible values and meanings of inhatr are now left to the implementation.

ATT_ISR	Attach Interrupt Service Routine (Static API)
cre_isr	Create Interrupt Service Routine
acre_isr	Create Interrupt Service Routine (ID Number Automatic Assignment)

[Static API]

ATT_ISR ({ ATR isratr, VP_INT exinf, INTNO intno, FP isr }) ;

[C Language API]

ER ercd = cre_isr (ID isrid, T_CISR *pk_cisr) ;
ER_ID isrid = acre_isr (T_CISR *pk_cisr) ;

[Parameter]

ID	isrid	ID number of the interrupt service routine to be created (only cre_isr)
T_CISR *	pk_cisr	Pointer to the packet containing the interrupt service routine creation information (in ATT_ISR, packet contents must be directly specified.)

pk_cisr includes (T_CISR type)

ATR	isratr	Interrupt service routine attribute
VP_INT	exinf	Interrupt service routine extended information
INTNO	intno	Interrupt number to which the interrupt service routine is to be attached
FP	isr	Interrupt service routine start address

(Other implementation specific information may be added.)

[Return Parameter]

cre_isr:

ER	ercd	E_OK for normal completion or error code

acre_isr:

ER_ID	isrid	ID number (positive value) of the created interrupt service routine or error code

[Error Code]

E_ID	Invalid ID number (isrid is invalid or unusable; only cre_isr)
E_NOID	No ID number available (there is no interrupt service routine ID assignable; only acre_isr)
E_RSATR	Reserved attribute (isratr is invalid or unusable)
E_PAR	Parameter error (pk_cisr, intno, or isr is invalid)
E_OBJ	Object state error (interrupt service routine is already registered; only cre_isr)

[Functional Description]

These service calls create an interrupt service routine with an ID number specified by isrid based on the information contained in the packet pointed to by pk_cisr. isratr is the attribute of the interrupt service routine. exinf is the extended information passed as a parameter to the interrupt service routine when it is called. intno is the number of the interrupt associated with the interrupt service routine. isr is the start address of interrupt service routine.

ATT_ISR is used to attach an interrupt service routine without assigning isrid. The interrupt service routines specified in this way have no ID numbers. In ATT_ISR, isratr is a preprocessor constant expression parameter. intno is an integer parameter without automatic assignment.

acre_isr assigns an interrupt service routine ID from the pool of unassigned interrupt service routine IDs and returns the assigned interrupt service routine ID.

isratr can be specified as (TA_HLNG || TA_ASM). If TA_HLNG (= 0x00) is specified, the interrupt service routine is called through the C language interface. If TA_ASM (= 0x01) is specified, the interrupt service routine is called through an assembly language interface.

[Standard Profile]

The Standard Profile does not require support for DEF_INH if the implementation supports ATT_ISR. In this case, the Standard Profile does not require support for when TA_ASM is specified in isratr.

[Supplemental Information]

Multiple interrupt service routines may be attached to the same interrupt number. See Section 3.3.2 for information on how to handle multiple interrupt service routines attached to the same interrupt number.

[Differences from the µITRON3.0 Specification]

This service call has been newly added.

del_isr Delete Interrupt Service Routine

[C Language API]

```
ER ercd = del_isr ( ID isrid ) ;
```

[Parameter]

ID	isrid	ID number of the interrupt service routine to be deleted

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_ID	Invalid ID number (isrid is invalid or unusable)
E_NOEXS	Non-existent object (specified interrupt service routine is not registered)

[Functional Description]

This service all deletes the interrupt service routine specified by isrid.

[Supplemental Information]

Interrupt service routines attached through ATT_ISR cannot be deleted with this service call because they do not have ID numbers.

[Differences from the μITRON3.0 Specification]

This service call has been newly added.

ref_isr Reference Interrupt Service Routine State

[C Language API]

ER ercd = ref_isr (ID isrid, T_RISR *pk_risr) ;

[Parameter]

ID	isrid	ID number of the interrupt service routine to be referenced
T_RISR *	pk_risr	Pointer a packet returning the interrupt service routine state

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

pk_risr includes (T_RISR type)

(Implementation-specific information)

[Error Code]

E_ID	Invalid ID number (isrid is invalid or unusable)
E_NOEXS	Non-existent object (specified interrupt service routine is not registered)
E_PAR	Parameter error (pk_risr is invalid)

[Functional Description]

The service call references the state of the interrupt service routine specified by isrid. The state of the interrupt service routine is returned through the packet pointed to by pk_risr. The specific information returned is implementation-defined.

[Differences from the μITRON3.0 Specification]

This service call has been newly added.

dis_int Disable Interrupt

[C Language API]

ER ercd = dis_int (INTNO intno) ;

[Parameter]

INTNO	intno	Interrupt number to be disabled

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_PAR	Parameter error (intno is invalid)

[Functional Description]

This service call disables the interrupt specified by intno. The specific meaning of intno is implementation-defined. In typical implementations, intno corresponds to the interrupt request line to the IRC.

[Supplemental Information]

This service call is intended to control the IRC. This service call does not transition the system to the CPU locked state nor does it transition the system to the dispatching disabled state. Therefore, dispatching still occurs even if all interrupts are disabled due to this service call. In addition, if interrupts are disabled, they remain disabled after task dispatching.

[Differences from the μITRON3.0 Specification]

Because this service call is intended to control the IRC, the meaning of intno is defined more strictly than in the μITRON3.0 Specification. The data type of intno has been changed from UINT to INTNO.

ena_int Enable Interrupt

[C Language API]

ER ercd = ena_int (INTNO intno) ;

[Parameter]

INTNO	intno	Interrupt number to be enabled

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_PAR	Parameter error (intno is invalid)

[Functional Description]

This service call enables the interrupt specified by intno. The specific meaning of intno is implementation-defined. In typical implementations, intno corresponds to the interrupt request line to the IRC.

[Supplemental Information]

This service call is intended to control the IRC. This service call does not transition the system to the CPU unlocked state nor does it transition the system to the dispatching enabled state. Therefore, this service call does not necessarily result in a state where interrupts will be accepted by the processor.

[Differences from the µITRON3.0 Specification]

Because this service call is intended to control the IRC, the meaning of intno is defined more strictly than in the µITRON3.0 Specification. The data type of intno has been changed from UINT to INTNO.

chg_ixx Change Interrupt Mask

[C Language API]

ER ercd = chg_ixx (IXXXX ixxxx) ;

[Parameter]

IXXXX	ixxxx	Interrupt mask desired

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_PAR	Parameter error (ixxxx is invalid)

[Functional Description]

This service call changes the processor's interrupt mask (also referred to as interrupt level or interrupt priority) to the value specified by ixxxx.

The xx portion of the service call name and the xxxx portion of the parameter name are implementation-defined and should be appropriate character strings for the target processor's architecture.

Depending on the value specified by ixxxx, this service call may cause the transition between the CPU locked state and the CPU unlocked state and/or the transition between the dispatching disabled state and the dispatching enabled state. The value causing these transitions and the transition caused by this service call are implementation-defined.

[Supplemental Information]

In implementations where the CPU state is managed with the interrupt mask, changing the interrupt mask may cause the transition between the CPU states or the transition between the dispatching states. In implementations where these states are managed by a combination of the interrupt mask and a variable, the variable's value must be updated to reflect the change in the interrupt mask.

[Differences from the μITRON3.0 Specification]

The data type for ixxxx has been change from UINT to IXXXX.

get_ixx Reference Interrupt Mask

[C Language API]

ER ercd = get_ixx (IXXXX *p_ixxxx) ;

[Parameter]

None

[Return Parameter]

ER	ercd	E_OK for normal completion or error code
IXXXX	ixxxx	Current interrupt mask

[Error Code]

E_PAR	Parameter error (p_ixxxx is invalid)

[Functional Description]

This service call references the processor's interrupt mask (also referred to as interrupt level or interrupt priority) and returns it through ixxxx.

The xx portion of the service call name and the xxxx portion of the parameter name are implementation-defined and should be appropriate character strings for the target processor's architecture.

[Differences from the µITRON3.0 Specification]

The name of this service call has been changed from ref_ixx to get_ixx. The data type for ixxxx has been change from UINT to IXXXX.

4.10 Service Call Management Functions

Service call management functions provide definition and invocation of extended service calls. The ability to invoke extended service calls may also be used to invoke standard service calls.

An extended service call is a function that allows the invocation of another module when the entire system is not linked to a single module. When an extended service call is invoked, the extended service call routine defined by the application is called.

The format to write an extended service call routine in the C language is shown below:

```
ER_UINT svcrtn ( VP_INT par1, VP_INT par2, ...)
{
        /* Body of the extended service call routine */
}
```

Only the necessary parameters for the extended service call routine (par1, par2, and so on) may be specified. There may be an implementation-defined limit on the number of parameters for extended service calls. However, at least one parameter must be supported.

The following data type packets are used for defining extended service calls:

```
typedef struct t_dsvc {
        ATR     svcatr ;    /* Extended service call attribute */
        FP      svcrtn ;    /* Extended service call routine start
                               address */
        /* Other implementation specific fields may be added. */
} T_DSVC ;
```

The following represents the function codes for service call management service calls. cal_svc has no function code.

TFN_DEF_SVC –0x6d Function code of def_svc

[Standard Profile]

The Standard Profile does not require support for service call management functions.

[Supplemental Information]

The contexts and states under which extended service call routines execute are summarized as follows:

- An extended service call routine executes in its own independent context determined by the extended service call and by the context from which the extended service call is invoked (see Section 3.5.1). The context in which an extended service call routine executes is classified as task contexts when the invoking context is classified as task contexts. It is classified as non-task contexts when the invoking context is classified as non-task contexts (See Section 3.5.2).
- The precedence of extended service call routines is higher than the precedence of the

processing unit that invokes the extended service calls and is lower than the precedence of any processing unit that has a higher precedence than the invoking processing unit (see Section 3.5.3).

- The start of and the return from extended service call routines do not change the CPU state and the dispatching state (See Sections 3.5.4 and 3.5.5).
- Executing extended service call routines with task exceptions disabled is implementation-defined (see Section 4.3).

[Differences from the μITRON3.0 Specification]

The category of service call management functions has been newly added.

The terms extended SVC and extended SVC handler have been changed to extended service call and extended service call routine, respectively. The contexts and states under which extended service call routines execute is more strictly defined compared to the μITRON3.0 Specification.

DEF_SVC Define Extended Service Call (Static API)
def_svc Define Extended Service Call

[Static API]

DEF_SVC (FN fncd, { ATR svcatr, FP svcrtn }) ;

[C Language API]

ER ercd = def_svc (FN fncd, T_DSVC *pk_dsvc) ;

[Parameter]

FN	fncd	Function code of the extended service call to be defined
T_DSVC *	pk_dsvc	Pointer to the packet containing the extended service call definition information (in DEF_SVC, packet contents must be directly specified.)

pk_dsvc includes (T_DSVC type)

ATR	svcatr	Extended service call attribute
FP	svcrtn	Extended service call routine start address

(Other implementation specific information may be added.)

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_RSATR	Reserved attribute (svcatr is invalid or unusable)
E_PAR	Parameter error (fncd, pk_dsvc, or svcatr is invalid)

[Functional Description]

This service call defines an extended service call for the function code specified by fncd based on the information contained in the packet pointed to by pk_dsvc. svcatr is the attribute of the extended service call. svcrtn is the start address of the extended service call routine.

In DEF_SVC, fncd is an integer parameter without automatic assignment. svcatr is a preprocessor constant expression parameter.

This service call and this static API can define an extended service call with a positive value of fncd. If a negative value is specified in fncd, an E_PAR error is reported.

If pk_dsvc is NULL (= 0), the extended service call currently defined is released and the extended service call becomes undefined. When a new extended service call is defined over top of an old one, the old one is released and the new takes its place.

svcatr can be specified as (TA_HLNG || TA_ASM). If TA_HLNG (= 0x00) is specified, the extended service call routine is called through the C language interface. If TA_ASM (= 0x01) is specified, the extended service call routine is called through an assembly language interface.

[Differences from the μITRON3.0 Specification]

The name of the parameter has been changed from svchdr to svcrtn.

cal_svc Invoke Service Call

[C Language API]

```
ER_UINT ercd = cal_svc ( FN fncd, VP_INT par1, VP_INT par2,
                         ...) ;
```

[Parameter]

FN	fncd	Function code of the service call to be invoked
VP_INT	par1	The first parameter of the service call
VP_INT	par2	The second parameter of the service call
...	...	(up to the necessary number of parameters)

[Return Parameter]

ER_UINT	ercd	The service call's return value

[Error Code]

E_RSFN	Reserved function code (fncd is invalid or unusable)

[Functional Description]

This service call invokes the service call specified by fncd with the parameters par1, par2, and so on, and returns the return value of the invoked service call.

There may be an implementation-defined limit greater than or equal to 1 on the number of parameters that can be passed to the service call. If the service call's parameters are not of VP_INT type, this service call converts the parameters to the appropriate data types while preserving their values. If the service call's return value is of ER, BOOL, or ER_BOOL type, this service call converts the return value to ER_UINT type while preserving its value.

In addition to an extended service call, allowing this service call to invoke a standard service call is implementation-defined. If this service call cannot invoke a standard service call, it returns an E_RSFN error.

[Supplemental Information]

Standard service calls are distinguished from extended service calls because the former have negative function codes. Since cal_svc does not have a function code, cal_svc cannot be used to invoke itself.

[Differences from the µITRON3.0 Specification]

This service call has been newly added.

4.11 System Configuration Management Functions

System configuration management functions include the ability to define a CPU exception handler, to reference the system configuration and version information, and to define an initialization routine. The initialization routine executes during system initialization. See Section 3.7 for the timing and contexts of initialization routine execution.

The following data types are used for system configuration management functions:

EXCNO	CPU exception handler number

The format to write a CPU exception handler is implementation-defined.

When calling an initialization routine, the extended information (exinf) of the initialization routine is passed as a parameter. The format to write an initialization routine in the C language is shown below:

```
void inirtn ( VP_INT exinf )
{
        /* Body of the initialization routine */
}
```

The following data type packets are defined for defining CPU exception handlers and for referencing the configuration and version information.

```
typedef struct t_dexc {
        ATR     excatr ;    /* CPU exception handler attribute */
        FP      exchdr ;    /* CPU exception handler start address */
        /* Other implementation specific fields may be added. */
} T_DEXC ;

typedef struct t_rcfg {
        /* Implementation specific fields */
} T_RCFG ;

typedef struct t_rver {
        UH      maker ;     /* Kernel maker's code */
        UH      prid ;      /* Identification number of the kernel */
        UH      spver ;     /* Version number of the ITRON
                               Specification */
        UH      prver ;     /* Version number of the kernel */
        UH      prno[4] ;   /* Management information of the kernel
                               product */
} T_RVER ;
```

The following represents the function codes for the system configuration management service calls:

TFN_DEF_EXC	–0x6e	Function code of def_exc
TFN_REF_CFG	–0x6f	Function code of ref_cfg
TFN_REF_VER	–0x70	Function code of ref_ver

[Standard Profile]

The Standard Profile requires support for the static API defining an CPU exception handler (DEF_EXC) and the static API defining an initialization routine (ATT_INI).

[Supplemental Information]

The contexts and states under which CPU exception handlers execute are summarized as follows:

- The service calls that can be invoked from within CPU exception handlers are implementation-defined (see Section 3.4.2).
- A CPU exception handler executes in its own independent context determined by the CPU exception and by the context in which the CPU exception occurred (see Section 3.5.1). When a CPU exception occurs in task contexts, whether the CPU exception handler executes in task contexts or in non-task contexts is implementation-defined. When a CPU exception occurs in non-task contexts, the CPU exception handler executes in non-task contexts (see Section 3.5.2).
- The precedence of CPU exception handlers is higher than the precedence of the processing unit where the CPU exception occurs and higher than the precedence of the dispatcher (see Section 3.5.3).
- The start of and the return from CPU exception handlers do not change the CPU state and the dispatching state. When the CPU state or the dispatching state is changed in CPU exception handlers, they should be returned to their previous states before returning from the CPU exception handlers (see Sections 3.5.4 and 3.5.5).

DEF_EXC Define CPU Exception Handler (Static API) [S]
def_exc Define CPU Exception Handler

[Static API]

DEF_EXC (EXCNO excno, { ATR excatr, FP exchdr }) ;

[C Language API]

ER ercd = def_exc (EXCNO excno, T_DEXC *pk_dexc) ;

[Parameter]

EXCNO	excno	CPU exception handler number to be defined
T_DEXC *	pk_dexc	Pointer to the packet containing the CPU exception handler definition information (in DEF_EXC, packet contents must be directly specified.)

pk_dexc includes (T_DEXC type)

ATR	excatr	CPU exception handler attribute
FP	exchdr	CPU exception handler start address

(Other implementation specific information may be added.)

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

[Error Code]

E_RSATR	Reserved attribute (excatr is invalid or unusable)
E_PAR	Parameter error (excno, pk_dexc, and exchdr is invalid)

[Functional Description]

This service call assigns a CPU exception handler to the CPU exception handler number specified by excno based on the information contained in the packet pointed to by pk_dexc. excatr is the attribute of CPU exception handler attribute. exchdr is the start address of the CPU exception handler.

In DEF_EXC, excno is an integer parameter without automatic assignment. excatr is a preprocessor constant expression parameter.

The specific meaning of excno is implementation-defined, but it corresponds to the processor's exception in typical implementations.

If pk_dexc is NULL (= 0), the CPU exception handler currently defined is released. When a new CPU exception handler is defined over top of an old one, the old one is released and the new takes its place.

The possible values and meanings of excatr are implementation-defined.

[Differences from the μITRON3.0 Specification]

This service call is now specified for defining a CPU exception handler. The object number for identifying a CPU exception handler is now the CPU exception handler number (excno) of EXCNO type. The possible values and meanings of excatr are now left to the implementation.

ref_cfg Reference Configuration Information

[C Language API]

ER ercd = ref_cfg (T_RCFG *pk_rcfg) ;

[Parameter]

T_RCFG *	pk_rcfg	Pointer to the packet returning the configuration information

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

pk_rcfg includes (T_RCFG type)

(Implementation-specific information)

[Error Code]

E_PAR	Parameter error (pk_rcfg is invalid)

[Functional Description]

This service call references the static information and configuration information of the system. The information is returned through the packet pointed to by pk_rcfg. The specific information referenced is implementation-defined.

[Supplemental Information]

Possible information that may be referenced by this service call includes: the kernel configuration constants, the range of ID numbers for each object, overview of the memory map, available memory size, information on peripheral chips and I/O devices, and time unit and precision of the data types to specify the time.

ref_ver Reference Version Information

[C Language API]

```
ER ercd = ref_ver ( T_RVER *pk_rver ) ;
```

[Parameter]

T_RVER *	pk_rver	Pointer to the packet returning the version information

[Return Parameter]

ER	ercd	E_OK for normal completion or error code

pk_rver includes (T_RVER type)

UH	maker	Kernel maker's code
UH	prid	Identification number of the kernel
UH	spver	Version number of the ITRON Specification
UH	prver	Version number of the kernel
UH	prno[4]	Management information of the kernel product

[Error Code]

E_PAR	Parameter error (pk_rver is invalid)

[Functional Description]

This service call references the version information of the kernel. The information is returned through the packet pointed to by pk_rver. Specifically, the following information can be referenced.

maker is the code that represents the kernel maker. See Section 5.4 for definitions of maker codes.

prid is the number for identifying the kernel. The kernel maker can assign values to prid. A particular kernel implementation should be uniquely identified by the combination of maker and prid codes.

The upper four bits of spver identify the type of the TRON Specification, and the lower 12 bits indicate the version number of the specification. The upper four bits of spver are assigned as follows:

0x0	Common specification for TRON (such as TAD)
0x1	ITRON Specifications (ITRON1, ITRON2)
0x2	BTRON Specifications
0x3	CTRON Specifications
0x5	μITRON Specifications (μITRON2.0, μITRON3.0, μITRON4.0)
0x6	μBTRON Specifications

The lower 12 bits of spver represent the upper 3 digits of the specification version number. The upper 3 digits of the specification version number are represented in binary coded decimal (BCD) format and each digit is 4 bit long. Version numbers for draft specifications or specifications under discussion can include an alphabet letter. In this case, the letter is interpreted as a hexadecimal number. See Section 5.3 for further description on version numbers of the ITRON Specification.

prver is the version number of the particular kernel implementation. The kernel maker can assign values to prver.

prno is a return parameter that may contain the kernel product's management information, product number, and others. The kernel maker determines its definition.

[Supplemental Information]

As an example, the value of spver for a kernel conformant to the μITRON4.0 Specification Ver.4.02.10 is 0x5402, and its value for a kernel conformant to Ver. 4.A1.01 is 0x54A1. This example shows that a newer version of the specification does not always have a larger value of spver when a draft specification is involved.

The returned information except prno can be referenced with the kernel configuration macros: TKERNEL_MAKER, TKERNEL_PRID, TKERNEL_SPVER, and TKERNEL_PRVER.

[Differences from the μITRON3.0 Specification]

The name of the service call has been changed from get_ver to ref_ver. Referencing the CPU information and the variation descriptor have been removed. The specification of the prver format has been removed. The name of the return parameter has been changed from id to prid.

[Rationale]

The values stored in spver include only the upper 3 digits of the specification version number and do not include the remaining digits. This is because the remaining digits only refer to the notation of the specification and not the contents.

ATT_INI Attach Initialization Routine (Static API) [S]

[Static API]

ATT_INI ({ ATR iniatr, VP_INT exinf, FP inirtn }) ;

[Parameter]

ATR	iniatr	Initialization routine attribute
VP_INT	exinf	Initialization routine extended information
FP	inirtn	Initialization routine start address

(Other implementation specific information may be added.)

[Functional Description]

This static API registers an initialization routine based on the specified parameters. iniatr is the attribute of the initialization routine. exinf is the extended information passed as a parameter to the initialization routine. inirtn is the start address of the initialization routine.

In ATT_INI, iniatr is a preprocessor constant expression parameter.

The registered initialization routine is executed as a part of the processing of the static APIs during system initialization. See Section 3.7 for a detailed description of this process.

iniatr can be specified as (TA_HLNG || TA_ASM). If TA_HLNG (= 0x00) is specified, the initialization routine is called through the C language interface. If TA_ASM (= 0x01) is specified, the initialization routine is called through an assembly language interface.

[Standard Profile]

The Standard Profile does not require support for when TA_ASM is specified in iniatr.

[Supplemental Information]

The system configuration file can include more than one ATT_INI. See Section 3.7 for the execution order of the initialization routines when more than one ATT_INI are described.

[Differences from the μITRON3.0 Specification]

This static API has been newly added.

Chapter 5 Additional Specifications

5.1 The Specification Requirements for the µITRON4.0 Specification

5.1.1 Basic Concept

The µITRON Specifications are specifications which are based on a loose standardization concept. It emphasizes applicability to a wide range of hardwares and applications rather than portability of application programs, and aims at standardization for the education of software engineers. Therefore, as long as the OS specification meets the minimum requirements of a real-time kernel, the realization of the functionality defined in this specification and the addition of extended functionalities are left to the implementation.

Specifically, the following conditions must be satisfied for the implementation of the µITRON4.0 Specification.

(a) It must have the minimum functionalities that are required to satisfy the µITRON4.0 Specification (see Section 5.1.2).

(b) If it contains functionalities similar to those described in the µITRON4.0 Specification, the functionality specifications must match the µITRON4.0 Specification. However, if the implementation does not provide a configurator, conforming to the static API specification of the µITRON4.0 Specification is not necessary.

(c) If it contains functionalities not specified by the µITRON4.0 Specification, the functionality specifications must satisfy the conditions for implementation-dependent extensions specified by the µITRON4.0 Specification. However, if the implementation supports several sets of APIs, this condition is not applied to sets of APIs other than the µITRON4.0 Specification APIs.

If the implementation provides subsetting of service call functionalities or functionality restrictions, or if it has special implementation functions that are not specified by the µITRON4.0 Specification, the product manual must contain the description of the implementation for clarification.

The profile rule defines the minimum function requirements that must be satisfied by the kernel for the portability of application programs that are written in a high-level language. In order for an implementation based on the µITRON4.0 Specification to conform to a certain profile rule, it must have all the functionalities specified by the profile, and it must agree with all the rules related to the profile. It can contain functionalities that are not included by the profile and implementation-specific extensions. However, application programs that are written to operate using only the functionalities included in the profile must operate without modification.

Moreover, when embedding the implemented kernel to an application, embedding only the functions needed by the application is possible.

[Standard Profile]

The Standard Profile is one of the profile specifications of the µITRON4.0 Specification.

[Supplemental Information]

The conditions under which an implementation satisfies the µITRON4.0 Specification is illustrated by the following example. If the implementation has semaphore functions, the names and functionalities of the service calls, the types, orders, and names of the parameters and return parameters, and the types and names of main error codes must all agree with the semaphore functions that are specified by the µITRON4.0 Specification. In this case, subsetting of service call functions is permitted at the cost of portability of application programs. If the implementation adds a functionality that is not specified by the µITRON4.0 Specification (like counting semaphores with priority inheritance), the functionality definition is freely decided by the implementation. Moreover, in situations where subsets or implementation-specific extensions are made, adding and/or deleting parameters and return parameters is permitted.

The conditions of this section do not specify a kernel configured for a particular application. When a kernel conforming to the Standard Profile is embedded to an application, the kernel functionalities may be limited to those functions needed by the application and the range of ID numbers and priorities may be limited.

5.1.2 Minimum Required Functionalities

The minimum functionalities that are required to satisfy the µITRON4.0 Specification are as follows:

(a) Creation of tasks. The task must at least be able to be in the RUNNING state, the READY state, and the DORMANT state.

(b) Task scheduling conforming to the µITRON4.0 Specification scheduling rule. However, restricting the number of tasks to one for each priority level or restricting the priority level to only one is allowed.

(c) Registration of interrupt handlers (or interrupt service routines).

(d) A method to activate tasks (changing the state from the DORMANT state to the READY state) from tasks and interrupt handlers (or interrupt service routines).

(e) A method for a task to terminate itself (changing the state from the READY state to the DORMANT state).

[Supplemental Information]

As an example, the minimum functionalities above can be satisfied if the implementation provides the service calls and static APIs below, and if its task scheduling rule follows the specification.

CRE_TSK	create task (static API)
act_tsk / iact_tsk	activate task
ext_tsk	terminate invoking task
DEF_INH	define interrupt handler (static API)

In this case, defining an interrupt handler (DEF_INH) can be replaced by attaching an interrupt service routine (ATT_ISR). If a configurator is not provided, providing equivalent methods with the static APIs instead of the static APIs conforming to the specification is sufficient. Also act_tsk and iact_tsk do not have to support queuing of activation requests. ext_tsk can be replaced by a return from the main routine.

[Differences from the µITRON3.0 Specification]

The minimum set of states for a task is changed from the RUNNING state, the READY state, and the WAITING state to the RUNNING state, the READY state, and the DORMANT state. The service calls required to be supported (level R) is not defined.

5.1.3 Extension of the µITRON4.0 Specification

When adding implementation-specific service calls to realize a new functionality which is not specified by the µITRON4.0 Specification, a “v” must be added in front of the name of the new service call. The names of the static API for implementation-specific functions are also based on this rule. However, the names for implementation-specific service calls that can be called from non-task contexts are exceptions to this rule (see Section 3.6.3). The value of the function code for any implementation-specific service call must be within the range provided.

When adding implementation-specific main error codes, the form of the name must be EV_*XXXXX* and the value of the main error code must be defined within the range provided. Also if there are implementation-specific data types, constants (except for error codes), and/or macros defined, identifying those that are not defined by the µITRON4.0 Specification by inserting a “V” into the name is recommended.

In the µITRON4.0 Specification, the constants that specify the object attributes and service call operational modes are assigned values that can be expressed in 8 bits. Also the constants that express the object states are assigned 8-bit values, with a few exceptions. The lower 8-bit values of the parameters or return parameters are reserved for future extensions of the ITRON Specifications. When assigning bit values to the implementation-specific constants for those parameters and return parameters, the bit values that cannot be used are the bits which are used by the constants defined by this specification and the reserved lowest 8 bits. The remaining values in the upper 8 bits should be used.

Also if there are rules that specify methods for implementation-specific extensions, such as the packet for object registration information and object reference information, these rules must be followed.

5.2 Automotive Control Profile

The Automotive Control Profile of the µITRON4.0 Specification is one of the µITRON4.0 Specification profile rules and is mainly targeted at automotive control applications. In order to realize the goal of reducing kernel overhead and memory usage, a subsetting of the specification functions and additional functions for reducing memory consumption are provided.

Compared to the Standard Profile, the Automotive Control Profile does not need to support the following functionalities:

- Service calls with timeouts
- Wait queues in task priority order
- The SUSPENDED state
- Task exception handling functions
- Mailboxes
- Fixed-sized memory pools
- Some other service calls

In order to reduce the memory usage, restricted task functions are added. Restricted tasks are tasks whose functionalities are restricted compared to conventional tasks. As long as the application does not depend on an E_NOSPT error returned when restricted functions are used, the application should behave the same way if the restricted tasks are replaced with conventional tasks. In this sense, the Automotive Control Profiles have a lower compatibility to the Standard Profile even if restricted task functions are added.

5.2.1 Restricted Tasks

By restricting some functionalities of tasks, a restricted task can share the same stack space with other restricted tasks. This reduces the memory area required for task stack. A restricted task differs from a conventional task as follows:

- A restricted task can not enter the WAITING state.

  When a restricted task invokes a service call that might enter the WAITING state, the behavior is undefined. When an error should be reported, an E_NOSPT error is returned.

- The priority of a restricted task cannot be changed.

  The behavior of changing a restricted task's priority by invoking chg_pri is undefined. When an error should be reported, an E_NOSPT error is returned.

- A restricted task cannot be terminated by a service call.

A restricted task can only be terminated by returning from the task's main routine. The behavior when a restricted task terminate itself by invoking ext_tsk and the behavior when a restricted task is forcibly terminated through ter_tsk are undefined. When an error should be reported, an E_NOSPT error is returned.

Whether the task is restricted or not is determined by the task attribute specified during task creation. Specifically, the task will be a restricted task if the task is created by specifying TA_RSTR (= 0x04) in the task attribute.

[Supplemental Information]

Specifying the task stack size, which is included in the task creation information, is also valid for a restricted task. For example, if the same stack area is shared by several restricted tasks of the same priority, setting the maximum value of each task's stack size to the size of the stack area allocated by the kernel is necessary. Like the Standard Profile, the Automotive Control Profile does not require support for when other values than NULL are specified as the start address of a task stack space.

5.2.2 Functionalities Included in the Automotive Control Profile

All the functionalities of the Automotive Control Profile except for the restricted task functions are included in the Standard Profile. The Automotive Control Profile must support the following static APIs and service calls.

(1) Task management functions

CRE_TSK	create task (static API)
act_tsk / iact_tsk	activate task
can_act	cancel task activation requests
ext_tsk	terminate invoking task
ter_tsk	terminate task
chg_pri	change task priority
get_pri	reference task priority

(2) Task dependent synchronization functions

slp_tsk	put task to sleep
wup_tsk / iwup_tsk	wakeup task
can_wup	cancel task wakeup requests
rel_wai / irel_wai	release task from waiting

(4) Synchronization and communication functions

Semaphores

CRE_SEM	create semaphore (static API)
sig_sem / isig_sem	release semaphore resource
wai_sem	acquire semaphore resource
pol_sem	acquire semaphore resource (polling)

Eventflags

CRE_FLG	create eventflag (static API)
set_flg / iset_flg	set eventflag
clr_flg	clear eventflag
wai_flg	wait for eventflag
pol_flg	wait for eventflag (polling)

Data queues

CRE_DTQ	create data queue (static API)
psnd_dtq / ipsnd_dtq	send to data queue (polling)
fsnd_dtq / ifsnd_dtq	forced send to data queue
rcv_dtq	receive from data queue
prcv_dtq	receive from data queue (polling)

(7) Time management functions

System time management

isig_tim	supply time tick

* If the kernel has a mechanism of updating the system time, isig_tim need not be supported.

Cyclic handlers

CRE_CYC	create cyclic handler (static API)
sta_cyc	start cyclic handler operation
stp_cyc	stop cyclic handler operation

(8) System state management functions

get_tid / iget_tid	reference task ID in the RUNNING state
loc_cpu / iloc_cpu	lock the CPU
unl_cpu / iunl_cpu	unlock the CPU
dis_dsp	disable dispatching
ena_dsp	enable dispatching
sns_ctx	reference contexts
sns_loc	reference CPU state
sns_dsp	reference dispatching state
sns_dpn	reference dispatch pending state

(9) Interrupt management functions

DEF_INH	define interrupt handler (static API)

* If ATT_ISR is supported, DEF_INH need not be supported.

(11) System configuration management functions

DEF_EXC	define CPU exception handler (static API)
ATT_INI	attach initialization routine (static API)

Among these static APIs or service calls, the functions that should be supported by the Automotive Control Profile but are restricted or extended compared to the Standard

Profile are as follows.

- CRE_TSK

  TA_RSTR (= 0x04) can be specified in the task attribute. When TA_RSTR is specified, a restricted task is created.

- CRE_SEM, CRE_FLG, CRE_DTQ

  The Automotive Control Profile does not require support for when TA_TPRI is specified in each object attribute.

- ext_tsk

  The behavior when invoked from restricted tasks is undefined. When an error should be reported, an E_NOSPT error is returned.

- ter_tsk, chg_pri

  The behavior when invoked with a restricted task is undefined. When an error should be reported, an E_NOSPT error is returned.

- slp_tsk, wai_sem, wai_flg, rcv_dtq

  The behavior when invoked from a restricted task is undefined. When an error should be reported, an E_NOSPT error is returned.

[Supplemental Information]

Within the Automotive Control Profile, the behavior when TA_TFIFO is specified as the eventflag attribute is the same as when TA_TPRI is specified. In addition, since the task cannot enter the sending waiting state for a data queue, specifying TA_TFIFO or TA_TPRI for the data queue attribute is meaningless. Therefore, the restriction that TA_TPRI cannot be specified for the eventflag attribute and the data queues attribute practically means that when TA_TPRI is specified, an error should be returned.

5.3 Version Number of the Specifications

The version number of the ITRON Specifications is in the following form:

Ver. *X.YY.ZZ* [.*WW*]

X represents the major version number of the ITRON Specifications. The numbers below are assigned to the kernel specifications:

1	ITRON1
2	ITRON2 or µITRON (Ver. 2.0)
3	µITRON3.0
4	µITRON4.0

YY indicates the version number of the updated specification when modifications or additions are made to its contents. Once the specification is published, *YY* is updated to *YY* = 00, 01, 02, and so on for each version of the specification. For draft specifications or specifications under discussion, on the other hand, one of the letters in *YY* should be 'A', 'B', or 'C'.

The $X.YY$ portion in the version number can be referenced through the kernel configuration macro TKERNEL_SPVER and through the return parameter spver of ref_ver service call. If YY contains 'A', 'B', or 'C', the hexadecimal representation of 'A', 'B', or 'C' is used, respectively.

ZZ is a number identifying the version relating to the specification notation. When structural changes are made to the specification document or chapters, or when typographical errors are corrected, ZZ is updated to ZZ = 00, 01, 02, and so on.

WW may be used for minor classifications on notations in the specification document. If WW is omitted, WW is regarded as 00.

5.4 Maker Codes

The TRON Association assigns the maker codes referenced through the kernel configuration macro TKERNEL_MAKER and through the return parameter maker of ref_ver service call.

At the time of the publication of this specification document, the following maker codes are assigned:

0x0000	No maker code (such as experimental systems)
0x0001	University of Tokyo
0x0008	Individuals (or personal businesses)
0x0009	FUJITSU LIMITED
0x000a	Hitachi, Ltd.
0x000b	Matsushita Electric Industrial Co., Ltd.
0x000c	Mitsubishi Electric Corporation
0x000d	NEC Corporation
0x000e	Oki Electric Industry Company, Limited
0x000f	Toshiba Corporation
0x0010	ALPS ELECTRIC Co., Ltd.
0x0011	WACOM Co., Ltd.
0x0012	Personal Media Corporation
0x0013	Sony Corporation
0x0014	Motorola, Inc.
0x0015	National Semiconductor Corporation
0x0101	OMRON Corporation
0x0102	Seiko Precision Inc.
0x0103	System Algo Co., Ltd.
0x0104	TOKYO COMPUTER SERVICE Co., Ltd.
0x0105	Yamaha Corporation
0x0106	MORSON JAPAN

0x0107	Toshiba Information Systems (JAPAN) Corporation
0x0108	MiSPO Co., Ltd.
0x0109	Three Ace Computer Corporation
0x010a	FIRMWARE SYSTEMS Inc.
0x010b	eSOL Co., Ltd.
0x010c	U S Software Corporation
0x010d	ACCESS CO., LTD.
0x010e	FUJITSU DEVICES INC.
0x010f	Accelerated Technology Incorporated
0x0110	ELMIC SYSTEMS, INC.
0x0111	FJB Web Technology Ltd.
0x0112	A. I. Corporation

For the kernels implemented by individuals (or personal businesses), 0x0008 is used as the maker code. For further identification of the kernel implementor, unique values are assigned to each individual in the upper 8 bits of the identification number of the kernel, which can be referenced through the kernel configuration macro TKERNEL_PRID and through the return parameter prid of ref_ver service call.

Chapter 6 Appendix

6.1 Conditions for Using the Specification and the Specification Document

The conditions for using the µITRON4.0 Specification and its specification document are as follows:

Conditions for Using the Specification

The µITRON4.0 Specification is an open specification. Anyone may freely develop, use, distribute, and sell software that conforms to the µITRON4.0 Specification. There is no need to pay a license fee or register to the ITRON Committee of the TRON Association.

However, the ITRON Committee of the TRON Association strongly recommends that the following statements (or statements with the same meaning) be included in the documentation of the software, such as the product manuals, conforming to the µITRON4.0 Specification:

- TRON is the abbreviation of "The Real-time Operating system Nucleus."
- ITRON is the abbreviation of "Industrial TRON."
- µITRON is the abbreviation of "Micro Industrial TRON."
- TRON, ITRON, and µITRON do not refer to any specific product or products.

The ITRON Committee of the TRON Association also recommends that the following statements (or statements with the same meaning) be included in the documentation of the software, such as the product manuals, conforming to the µITRON4.0 Specification:

> The µITRON4.0 Specifications is an open real-time kernel specification developed by the ITRON Committee of the TRON Association. The µITRON4.0 Specification document can be obtained from the ITRON Project web site (http://www.itron.gr.jp/).

If you receive permission to modify the specification document to create product manuals (described later), or if you register products to the ITRON-Specification Product Registration System (see Section 6.2), you are obliged to include the statements described above.

Conditions for Using the Specification Document

The copyright of the µITRON4.0 Specification document belongs to the ITRON Com-

Disclaimer

6.2 Maintenance of the Specification and Related Information

Maintenance of the ITRON Specifications and Contact Information

The ITRON Specifications and their documents are developed and maintained by the ITRON Committee of the TRON Association. Any questions regarding the specifications and their documents should be directed to the following:

ITRON Committee, TRON Association
Katsuta Building 5F
3-39, Mita 1-chome, Minato-ku,
Tokyo 108-0073, JAPAN
TEL: +81-3-3454-3191
FAX: +81-3-3454-3224

ITRON Project Web Site

The ITRON Committee of the TRON Association maintains the ITRON Project web site for distributing information regarding the ITRON Project and Specifications. Various ITRON Specifications and other documents are available at the web site, such as: introduction to the ITRON Project, the ITRON Newsletter, status of standardization activities, results of the survey on RTOS uses, list of products registered to the ITRON-Supplications Product Registration System, information on seminars and trade show participation, presentation materials used in lectures, and the list of the ITRON Committee members.

The URL of the ITRON Project Web Site is:

http://www.itron.gr.jp/

The ITRON Newsletter

The ITRON Committee of the TRON Association publishes the ITRON Newsletter bimonthly to widely distribute the latest information regarding the ITRON Project and the activities of the ITRON Committee. The ITRON Newsletter has both Japanese and English versions. Information regarding additions or corrections to the ITRON Specifications, and information regarding corrections to the books published by the ITRON Committee are notified with the ITRON Newsletter. The ITRON Newsletter is also used to introduce products, books, and documents related to the ITRON Specifications, and to notify the events such as seminars and trade shows.

The ITRON Newsletter is included in the TRONWARE magazine (only in Japanese) and the periodicals by the TRON Association. The ITRON Newsletter is also available at the ITRON Project web site.

ITRON-Specification Product Registration System

In order to promote the use and development of the ITRON Specifications, the ITRON Committee of the TRON Association provides the ITRON-Specification Product Registration System. The purpose of this system is to create and maintain a list of products developed by companies that conform to the ITRON Specifications and to promote the use of the ITRON Specifications and the conforming products. This system is different from the so-called certification system. It is not intended to certify registered products to be conformant to the ITRON Specifications.

The list of products registered to the system is available at the ITRON Project web site. Contact the ITRON Committee if you are interested in registering products that conform to the ITRON Specifications.

Reference documents

“THE TRON PROJECT” is published by the TRON Association as a reference for the entire TRON Project. This document includes an introduction to the activities of each TRON basic and application sub-project, the history of the TRON Project, and the list of the reference regarding the TRON Project.

For the latest information on the TRON Project, refer to “TRONWARE,” a TRON Project technical information magazine published bimonthly by Personal Media Corporation. For the research results of the TRON Project, refer to the proceedings of the annual TRON Project International Symposium.

The ITRON Committee of the TRON Association publishes the ITRON Specification Guidebook as a textbook regarding the ITRON Specifications.

> “ITRON Specification Guidebook 2,” supervised by Ken Sakamura, Personal Media Corporation, 1994 (ISBN4-89362-133-5).

“ITRON Specifications Guidebook 2” is based on the μITRON3.0 Specification and does not correspond to the μITRON4.0 Specification. However, the ITRON Committee is planning to publish an edition that corresponds to the μITRON4.0 Specification.

6.3 Background and Development Process of the Specification

Background and Development Process of the Specification

The ITRON Committee of the TRON Association started the μITRON4.0 Specification Study Group to develop the next generation μITRON Specification following the results of the Hard Real-Time Support Study Group (from November 1996 to March 1998) and of the RTOS Automotive Application Technical Committee (from June 1997 to March 1998). The μITRON4.0 Specification Study Group was an open group where anyone, including non-members of either the ITRON Committee or the TRON Association, was welcome to participate, thus promoting the involvement of active engineers from various fields of embedded system development.

The Kernel Specification Working Group established under the μITRON4.0 Specification Study Group developed the μITRON4.0 Specification. The Kernel Specification Working Group started the development in April 1998. It organized meetings once or twice a month until June 1999, when the official specification document was published. Email discussions were also conducted for the development.

The μITRON4.0 Specification also reflects the requirements and ideas derived from the following investigations: the ITRON TCP/IP API Specification by the Embedded TCP/IP Technical Committee, the JTRON2.0 Specification by Java Technology on ITRON-Specification OS Technical Committee, and investigations by the Device Driver Design Guideline Working Group of the μITRON4.0 Specification Study Group.

Member List of the ITRON Committee of the TRON Association (in alphabetical order)

John Cheuck (Metrowerks Co., Ltd.)

Shouichi Hachiya (Aplix Corporation)
Makoto Hirayama (Hewlett-Packard Japan, Ltd.)
Noboru Hirose (FIRMWARE SYSTEMS Inc.)
Shigeru Honma (Yamaha Corporation)
Katsuhiko Ishida (Hitachi, Ltd.)
Hidehiro Ishii (YDC Corporation)
Norihiko Ito (Nihon Cygnus Solutions)
Tomihisa Kamada (ACCESS Co., Ltd.)
Tatsuya Kamei (Mitsubishi Electric Corporation)
Kenji Kudou (FUJITSU DEVICES Inc.), Vice-Chair
Akira Matsui (Personal Media Corporation)
Hiroshi Monden (NEC Corporation)
Tetsuo Oe (Oki Electric Industry Company, Limited)
Ken Sakamura (University of Tokyo)
Kazuo Sato (Toshiba Information Systems (JAPAN) Corporation)
Tetsu Shibashita (Mentor Graphics Japan Co., Ltd.)
Hiroaki Takada (Toyohashi University of Technology), Secretary
Tetsuo Takagi (DENSO CREATE Inc.)
Tohru Takeuchi (TRON Association), Secretariat
Kiichiro Tamaru (Toshiba Corporation), Chair
Yasutaka Tsunakawa (Sony Corporation)
Yiroyuki Watanabe (Seiko Instruments Inc.)

Member List of the Kernel Specification Working Group of the µITRON4.0 Specification Study Group (in alphabetical order)

Yoshitaka Adachi (Matsushita Electric Industry Co., Ltd.)
Yoshihiko Aoki (Sanyo Engineering & Construction Inc.)
Shigemasa Asai (Aisin Seiki Co., Ltd.)
Akihito Chiba (NIPPON TELECOMMUNICATIONS CONSULTING Co., Ltd.)
Kazuhiro Ibuka (Motorola Japan Ltd.)
Jun’ichi Iijima, Secretery
Kazutoyo Inamitsu (FUJITSU DEVICES Inc.)
Katsuhiko Ishida (Hitachi, Ltd.)
Masanori Ishikawa (YDC Corporation)
Kazunori Isomoto (Mazda Motor Corporation)
Norihiko Ito (Nihon Cygnus Solutions)
Takanao Ito (Fuji Electric Mie Design Co., Ltd.)
Yoshihisa Iwaki (Honda R&D Co., Ltd.)
Shouichi Hachiya (Aplix Corporation)
Shin’ichi Hashimoto (ACCESS Co., Ltd.)

Osamu Higashihara (NEC Information Systems, Ltd.)
Hiroki Hihara (NEC Corporation)
Michitaro Horiuchi (ACCESS Co., Ltd.)
Hiroshi Kako (DENSO CORPORATION)
Hironori Kaneda (Data Technology Inc.)
Hiroshi Kawaguchi (HANAZUKA ELECTRIC INDUSTRY Co., Ltd.)
Tuyoshi Kodama (Alpine Information System Inc.)
Manabu Kobayakawa (Hitachi, Ltd.)
Masakazu Kobayashi (Hitachi ULSI Systems Co., Ltd.)
Yasuhiro Kobayashi (FUJITSU LIMITED)
Takahiro Kudo (Data Technology Inc.)
Kenji Kudou (FUJITSU DEVICES Inc.)
Tadakatsu Masaki (Matsushita Information Systems Research Laboratory Hiroshima Co., Ltd.)
Takayuki Matsunaga (Yazaki Corporation)
Atsushi Miki (Sumitomo Electric Industries, Ltd.)
Tetsuo Miyauchi (NEC Microcomputer Technology, Ltd.)
Hisaya Miyamoto (Toshiba Corporation)
Hisanori Miyazaki (MiSPO Co., Ltd.)
Kohei Mugitani (Sharp Corporation)
Hiroyuki Muraki (Mitsubishi Electric Semiconductor Systems Corporation)
Hiroyuki Nagasaku (CRESCO Ltd.)
Ryuichi Naito (Nippon Business Solution)
Yuiku Nakai (DENSAN Co., Ltd.)
Ken’ichi Nakamura (Nihon Cygnus Solutions)
Shigeki Nankaku (Mitsubishi Electric Corporation)
Tomo Onozawa (Aishin Seiki Co., Ltd.)
Masayuki Osajima (ACCESS Co., Ltd.)
Hideto Sakamoto (EST K.K.)
Koji Sato (Toyota Motor Corporation)
Shuji Sato (Toshiba Engineering Corporation)
Tsutomu Sawada (Erg Co., Ltd.), Secretary
Masanobu Shigeta (Fuji Denki Co., Ltd.)
Kazu Shimazaki (SENNET, Inc.)
Masahiro Shukuguchi (Mitsubishi Electric Micro-Computer Application Software Co., Ltd.)
Hideaki Suganuma (Toyota Motor Corporation)
Kenji Suganuma (DENSO CORPORATION)
Akihiko Sugimoto (Data Technology Inc.)
Hiroaki Takada (Toyohashi University of Technology), Secretary

Shuji Takanashi (Toshiba Corporation)
Yosuke Takano (NEC Corporation)
Tohru Takeuchi (TRON Association), Secretariat
Noriaki Tanaka (DENSO CREATE Inc.)
Kazuhiko Taoka (MiSPO Co., Ltd.)
Kazuhiro Terauchi (Toshiba Information Systems (JAPAN) Corporation)
Shin'ichi Tsunashima (ACCESS Co., Ltd.)
Naotaka Uehara (Casio)
Masakazu Uemura (Fuji Electric Mie Design)
Shinjiro Yamada (Hitachi, Ltd.)
Tatsuo Yamada (Motorola Japan Ltd.)
Masaru Yamanaka (Nihon Cygnus Solutions)
Akira Yokozawa (Toshiba Corporation)
Tomoaki Yoshida (Toshiba Corporation)
Miyoko Yoshimura (ERG. Co., Ltd)
Yukio Yoshino (Communication And Technology Systems, Inc.)
Masahiko Watanabe (Communication And Technology Systems, Inc.)

Contributors to the English Translation of the μITRON4.0 Specification

Eva Austria Barcelon (Toyohashi University of Technology)
Christopher G. Brown (U S Software)
Donald Dunstan (U S Software)
Tadahiro Fukaya (FIRMWARE SYSTEMS Inc.)
Shin'ichi Hashimoto (ACCESS Co., Ltd.)
Kazutoyo Inamitsu (FUJITSU DEVICES Inc.)
Kazuhiro Inaoka (Mitsubishi Electric Semiconductor Systems Corporation)
Takeshi Kaneko (A.I.Corporation)
Hiroyuki Kato (A.I.Corporation)
Tsutomu Kindaichi (ELMIC SYSTEMS, INC)
Isao Kubota (ERG Co., Ltd.)
Akira Matsui (Personal Media Corporation)
Koji Mugita (GRAPE SYSTEMS INC.)
Hiroyuki Muraki (Mitsubishi Electric Semiconductor Systems Corporation)
Kohichi Nakamoto (NEC Corporation)
Nicholas James Withcy (U S Software)
Takuya Nomura (Matsushita Electric Industrial Co., Ltd.)
Tatsuo Obata (A.I.Corporation)
Tsutomu Sawada (ERG Co., Ltd.)
Hiroaki Takada (Toyohashi University of Technology)

Tetsuo Takagi (DENSO CREATE Inc.)
Tohru Takeuchi (TRON Association)
Shinjiro Yamada (Hitachi, Ltd.)
Koichi Yasutake (Matsushita Electric Industrial Co., Ltd.)
Akira Yokozawa (Toshiba Corporation)

6.4 Version History

May 10, 1999	Ver. 4.A0.00	A draft version released for public comments
May 17, 1999	Ver. 4.A1.00	Unfinished portion completed
June 1, 1999	Ver. 4.B0.00	(Working Group internal version)
June 10, 1999	Ver. 4.B1.00	(Working Group internal version)
June 30, 1999	Ver. 4.00.00	Official release published

Chapter 7 References

7.1 Service Call List

(1) Task management functions

ER ercd = cre_tsk (ID tskid, T_CTSK *pk_ctsk) ;
ER_ID tskid = acre_tsk (T_CTSK *pk_ctsk) ;
ER ercd = del_tsk (ID tskid) ;
ER ercd = act_tsk (ID tskid) ;
ER ercd = iact_tsk (ID tskid) ;
ER_UINT actcnt = can_act (ID tskid) ;
ER ercd = sta_tsk (ID tskid, VP_INT stacd) ;
void ext_tsk ( ) ;
void exd_tsk ( ) ;
ER ercd = ter_tsk (ID tskid) ;
ER ercd = chg_pri (ID tskid, PRI tskpri) ;
ER ercd = get_pri (ID tskid, PRI *p_tskpri) ;
ER ercd = ref_tsk (ID tskid, T_RTSK *pk_rtsk) ;
ER ercd = ref_tst (ID tskid, T_RTST *pk_rtst) ;

(2) Task dependent synchronization functions

ER ercd = slp_tsk ( ) ;
ER ercd = tslp_tsk (TMO tmout) ;
ER ercd = wup_tsk (ID tskid) ;
ER ercd = iwup_tsk (ID tskid) ;
ER_UINT wupcnt = can_wup (ID tskid) ;
ER ercd = rel_wai (ID tskid) ;
ER ercd = irel_wai (ID tskid) ;
ER ercd = sus_tsk (ID tskid) ;
ER ercd = rsm_tsk (ID tskid) ;
ER ercd = frsm_tsk (ID tskid) ;
ER ercd = dly_tsk (RELTIM dlytim) ;

(3) Task exception handling functions

ER ercd = def_tex (ID tskid, T_DTEX *pk_dtex) ;
ER ercd = ras_tex (ID tskid, TEXPTN rasptn) ;
ER ercd = iras_tex (ID tskid, TEXPTN rasptn) ;
ER ercd = dis_tex ( ) ;
ER ercd = ena_tex ( ) ;
BOOL state = sns_tex ( ) ;
ER ercd = ref_tex (ID tskid, T_RTEX *pk_rtex) ;

(4) Synchronization and communication functions

Semaphores

```
ER ercd = cre_sem ( ID semid, T_CSEM *pk_csem ) ;
ER_ID semid = acre_sem ( T_CSEM *pk_csem ) ;
ER ercd = del_sem ( ID semid ) ;
ER ercd = sig_sem ( ID semid ) ;
ER ercd = isig_sem ( ID semid ) ;
ER ercd = wai_sem ( ID semid ) ;
ER ercd = pol_sem ( ID semid ) ;
ER ercd = twai_sem ( ID semid, TMO tmout ) ;
ER ercd = ref_sem ( ID semid, T_RSEM *pk_rsem ) ;
```

Eventflags

```
ER ercd = cre_flg ( ID flgid, T_CFLG *pk_cflg ) ;
ER_ID flgid = acre_flg ( T_CFLG *pk_cflg ) ;
ER ercd = del_flg ( ID flgid ) ;
ER ercd = set_flg ( ID flgid, FLGPTN setptn ) ;
ER ercd = iset_flg ( ID flgid, FLGPTN setptn ) ;
ER ercd = clr_flg ( ID flgid, FLGPTN clrptn ) ;
ER ercd = wai_flg ( ID flgid, FLGPTN waiptn, MODE wfmode,
                        FLGPTN *p_flgptn ) ;
ER ercd = pol_flg ( ID flgid, FLGPTN waiptn, MODE wfmode,
                        FLGPTN *p_flgptn ) ;
ER ercd = twai_flg ( ID flgid, FLGPTN waiptn, MODE wfmode,
                        FLGPTN *p_flgptn, TMO tmout ) ;
ER ercd = ref_flg ( ID flgid, T_RFLG *pk_rflg ) ;
```

Data queues

```
ER ercd = cre_dtq ( ID dtqid, T_CDTQ *pk_cdtq ) ;
ER_ID dtqid = acre_dtq ( T_CDTQ *pk_cdtq ) ;
ER ercd = del_dtq ( ID dtqid ) ;
ER ercd = snd_dtq ( ID dtqid, VP_INT data ) ;
ER ercd = psnd_dtq ( ID dtqid, VP_INT data ) ;
ER ercd = ipsnd_dtq ( ID dtqid, VP_INT data ) ;
ER ercd = tsnd_dtq ( ID dtqid, VP_INT data, TMO tmout ) ;
ER ercd = fsnd_dtq ( ID dtqid, VP_INT data ) ;
ER ercd = ifsnd_dtq ( ID dtqid, VP_INT data ) ;
ER ercd = rcv_dtq ( ID dtqid, VP_INT *p_data ) ;
ER ercd = prcv_dtq ( ID dtqid, VP_INT *p_data ) ;
ER ercd = trcv_dtq ( ID dtqid, VP_INT *p_data, TMO tmout ) ;
ER ercd = ref_dtq ( ID dtqid, T_RDTQ *pk_rdtq ) ;
```

Mailboxes

```
ER ercd = cre_mbx ( ID mbxid, T_CMBX *pk_cmbx ) ;
```

```
ER_ID mbxid = acre_mbx ( T_CMBX *pk_cmbx ) ;
ER ercd = del_mbx ( ID mbxid ) ;
ER ercd = snd_mbx ( ID mbxid, T_MSG *pk_msg ) ;
ER ercd = rcv_mbx ( ID mbxid, T_MSG **ppk_msg ) ;
ER ercd = prcv_mbx ( ID mbxid, T_MSG **ppk_msg ) ;
ER ercd = trcv_mbx ( ID mbxid, T_MSG **ppk_msg,
                        TMO tmout ) ;
ER ercd = ref_mbx ( ID mbxid, T_RMBX *pk_rmbx ) ;
```

(5) Extended synchronization and communication functions

Mutexes

```
ER ercd = cre_mtx ( ID mtxid, T_CMTX *pk_cmtx ) ;
ER_ID mtxid = acre_mtx ( T_CMTX *pk_cmtx ) ;
ER ercd = del_mtx ( ID mtxid ) ;
ER ercd = loc_mtx ( ID mtxid ) ;
ER ercd = ploc_mtx ( ID mtxid ) ;
ER ercd = tloc_mtx ( ID mtxid, TMO tmout ) ;
ER ercd = unl_mtx ( ID mtxid ) ;
ER ercd = ref_mtx ( ID mtxid, T_RMTX *pk_rmtx ) ;
```

Message buffers

```
ER ercd = cre_mbf ( ID mbfid, T_CMBF *pk_cmbf ) ;
ER_ID mbfid = acre_mbf ( T_CMBF *pk_cmbf ) ;
ER ercd = del_mbf ( ID mbfid ) ;
ER ercd = snd_mbf ( ID mbfid, VP msg, UINT msgsz ) ;
ER ercd = psnd_mbf ( ID mbfid, VP msg, UINT msgsz ) ;
ER ercd = tsnd_mbf ( ID mbfid, VP msg, UINT msgsz,
                        TMO tmout ) ;
ER_UINT msgsz = rcv_mbf ( ID mbfid, VP msg ) ;
ER_UINT msgsz = prcv_mbf ( ID mbfid, VP msg ) ;
ER_UINT msgsz = trcv_mbf ( ID mbfid, VP msg, TMO tmout ) ;
ER ercd = ref_mbf ( ID mbfid, T_RMBF *pk_rmbf ) ;
```

Rendezvous

```
ER ercd = cre_por ( ID porid, T_CPOR *pk_cpor ) ;
ER_ID porid = acre_por ( T_CPOR *pk_cpor ) ;
ER ercd = del_por ( ID porid ) ;
ER_UINT rmsgsz = cal_por ( ID porid, RDVPTN calptn, VP msg,
                        UINT cmsgsz ) ;
ER_UINT rmsgsz = tcal_por ( ID porid, RDVPTN calptn, VP msg,
                        UINT cmsgsz, TMO tmout ) ;
ER_UINT cmsgsz = acp_por ( ID porid, RDVPTN acpptn,
                        RDVNO *p_rdvno, VP msg ) ;
```

```
ER_UINT cmsgsz = pacp_por ( ID porid, RDVPTN acpptn,
                            RDVNO *p_rdvno, VP msg ) ;
ER_UINT cmsgsz = tacp_por ( ID porid, RDVPTN acpptn,
                            RDVNO *p_rdvno, VP msg, TMO tmout ) ;
ER ercd = fwd_por ( ID porid, RDVPTN calptn, RDVNO rdvno,
                            VP msg, UINT cmsgsz ) ;
ER ercd = rpl_rdv ( RDVNO rdvno, VP msg, UINT rmsgsz ) ;
ER ercd = ref_por ( ID porid, T_RPOR *pk_rpor ) ;
ER ercd = ref_rdv ( RDVNO rdvno, T_RRDV *pk_rrdv ) ;
```

(6) Memory pool management functions

Fixed-sized memory pools

```
ER ercd = cre_mpf ( ID mpfid, T_CMPF *pk_cmpf ) ;
ER_ID mpfid = acre_mpf ( T_CMPF *pk_cmpf ) ;
ER ercd = del_mpf ( ID mpfid ) ;
ER ercd = get_mpf ( ID mpfid, VP *p_blk ) ;
ER ercd = pget_mpf ( ID mpfid, VP *p_blk ) ;
ER ercd = tget_mpf ( ID mpfid, VP *p_blk, TMO tmout ) ;
ER ercd = rel_mpf ( ID mpfid, VP blk ) ;
ER ercd = ref_mpf ( ID mpfid, T_RMPF *pk_rmpf ) ;
```

Variable-sized memory pools

```
ER ercd = cre_mpl ( ID mplid, T_CMPL *pk_cmpl ) ;
ER_ID mplid = acre_mpl ( T_CMPL *pk_cmpl ) ;
ER ercd = del_mpl ( ID mplid ) ;
ER ercd = get_mpl ( ID mplid, UINT blksz, VP *p_blk ) ;
ER ercd = pget_mpl ( ID mplid, UINT blksz, VP *p_blk ) ;
ER ercd = tget_mpl ( ID mplid, UINT blksz, VP *p_blk,
                            TMO tmout ) ;
ER ercd = rel_mpl ( ID mplid, VP blk ) ;
ER ercd = ref_mpl ( ID mplid, T_RMPL *pk_rmpl ) ;
```

(7) Time management functions

System time management

```
ER ercd = set_tim ( SYSTIM *p_systim ) ;
ER ercd = get_tim ( SYSTIM *p_systim ) ;
ER ercd = isig_tim ( ) ;
```

Cyclic handlers

```
ER ercd = cre_cyc ( ID cycid, T_CCYC *pk_ccyc ) ;
ER_ID cycid = acre_cyc ( T_CCYC *pk_ccyc ) ;
ER ercd = del_cyc ( ID cycid ) ;
ER ercd = sta_cyc ( ID cycid ) ;
ER ercd = stp_cyc ( ID cycid ) ;
```

ER ercd = ref_cyc (ID cycid, T_RCYC *pk_rcyc) ;

Alarm handlers

ER ercd = cre_alm (ID almid, T_CALM *pk_calm) ;
ER_ID almid = acre_alm (T_CALM *pk_calm) ;
ER ercd = del_alm (ID almid) ;
ER ercd = sta_alm (ID almid, RELTIM almtim) ;
ER ercd = stp_alm (ID almid) ;
ER ercd = ref_alm (ID almid, T_RALM *pk_ralm) ;

Overrun handler

ER ercd = def_ovr (T_DOVR *pk_dovr) ;
ER ercd = sta_ovr (ID tskid, OVRTIM ovrtim) ;
ER ercd = stp_ovr (ID tskid) ;
ER ercd = ref_ovr (ID tskid, T_ROVR *pk_rovr) ;

(8) System state management functions

ER ercd = rot_rdq (PRI tskpri) ;
ER ercd = irot_rdq (PRI tskpri) ;
ER ercd = get_tid (ID *p_tskid) ;
ER ercd = iget_tid (ID *p_tskid) ;
ER ercd = loc_cpu ( ) ;
ER ercd = iloc_cpu ( ) ;
ER ercd = unl_cpu ( ) ;
ER ercd = iunl_cpu ( ) ;
ER ercd = dis_dsp ( ) ;
ER ercd = ena_dsp ( ) ;
BOOL state = sns_ctx ( ) ;
BOOL state = sns_loc ( ) ;
BOOL state = sns_dsp ( ) ;
BOOL state = sns_dpn ( ) ;
ER ercd = ref_sys (T_RSYS *pk_rsys) ;

(9) Interrupt management functions

ER ercd = def_inh (INHNO inhno, T_DINH *pk_dinh) ;
ER ercd = cre_isr (ID isrid, T_CISR *pk_cisr) ;
ER_ID isrid = acre_isr (T_CISR *pk_cisr) ;
ER ercd = del_isr (ID isrid) ;
ER ercd = ref_isr (ID isrid, T_RISR *pk_risr) ;
ER ercd = dis_int (INTNO intno) ;
ER ercd = ena_int (INTNO intno) ;
ER ercd = chg_i*xx* (I*XXXX* i*xxxx*) ;
ER ercd = get_i*xx* (I*XXXX* *p_i*xxxx*) ;

(10) Service call management functions

```
ER ercd = def_svc ( FN fncd, T_DSVC *pk_dsvc ) ;
ER_UINT ercd = cal_svc ( FN fncd, VP_INT par1, VP_INT par2,
                        ...) ;
```

(11) System configuration management functions

```
ER ercd = def_exc ( EXCNO excno, T_DEXC *pk_dexc ) ;
ER ercd = ref_cfg ( T_RCFG *pk_rcfg ) ;
ER ercd = ref_ver ( T_RVER *pk_rver ) ;
```

7.2 Static API List

(1) Task management functions

```
CRE_TSK ( ID tskid, { ATR tskatr, VP_INT exinf, FP task,
                        PRI itskpri, SIZE stksz, VP stk } ) ;
```

(3) Task exception handling functions

```
DEF_TEX ( ID tskid, { ATR texatr, FP texrtn } ) ;
```

(4) Synchronization and communication functions

```
CRE_SEM ( ID semid, { ATR sematr, UINT isemcnt,
                        UINT maxsem } ) ;
CRE_FLG ( ID flgid, { ATR flgatr, FLGPTN iflgptn } ) ;
CRE_DTQ ( ID dtqid, { ATR dtqatr, UINT dtqcnt, VP dtq } ) ;
CRE_MBX ( ID mbxid, { ATR mbxatr, PRI maxmpri,
                    VP mprihd } ) ;
```

(5) Extended synchronization and communication functions

```
CRE_MTX ( ID mtxid, { ATR mtxatr, PRI ceilpri } ) ;
CRE_MBF ( ID mbfid, { ATR mbfatr, UINT maxmsz, SIZE mbfsz,
                        VP mbf } ) ;
CRE_POR ( ID porid, { ATR poratr, UINT maxcmsz,
                        UINT maxrmsz } ) ;
```

(6) Memory pool management functions

```
CRE_MPF ( ID mpfid, { ATR mpfatr, UINT blkcnt, UINT blksz,
                        VP mpf } ) ;
CRE_MPL ( ID mplid, { ATR mplatr, SIZE mplsz, VP mpl } ) ;
```

(7) Time management functions

```
CRE_CYC ( ID cycid, { ATR cycatr, VP_INT exinf, FP cychdr,
                        RELTIM cyctim, RELTIM cycphs } ) ;
CRE_ALM ( ID almid, { ATR almatr, VP_INT exinf, FP almhdr } ) ;
DEF_OVR ( { ATR ovratr, FP ovrhdr } ) ;
```

(9) Interrupt management functions

DEF_INH (INHNO inhno, { ATR inhatr, FP inthdr }) ;
ATT_ISR ({ ATR isratr, VP_INT exinf, INTNO intno, FP isr }) ;

(10) Service call management functions

DEF_SVC (FN fncd, { ATR svcatr, FP svcrtn }) ;

(11) System configuration management functions

DEF_EXC (EXCNO excno, { ATR excatr, FP exchdr }) ;
ATT_INI ({ ATR iniatr, VP_INT exinf, FP inirtn }) ;

7.3 Static APIs and Service Calls in the Standard Profile

(1) Task management functions

CRE_TSK	Create Task (Static API)
act_tsk / iact_tsk	Activate Task
can_act	Cancel Task Activation Requests
ext_tsk	Terminate Invoking Task
ter_tsk	Terminate Task
chg_pri	Change Task Priority
get_pri	Reference Task Priority

(2) Task dependent synchronization functions

slp_tsk	Put Task to Sleep
tslp_tsk	Put Task to Sleep (with Timeout)
wup_tsk / iwup_tsk	Wakeup Task
can_wup	Cancel Task Wakeup Requests
rel_wai / irel_wai	Release Task from Waiting
sus_tsk	Suspend Task
rsm_tsk	Resume Suspended Task
frsm_tsk	Forcibly Resume Suspended Task
dly_tsk	Delay Task

(3) Task exception handling functions

DEF_TEX	Define Task Exception Handling Routine (Static API)
ras_tex / iras_tex	Raise Task Exception Handling
dis_tex	Disable Task Exceptions
ena_tex	Enable Task Exceptions
sns_tex	Reference Task Exception Handling State

(4) Reference Task Exception Handling State

Semaphores

CRE_SEM	Create Semaphore (Static API)
sig_sem / isig_sem	Release Semaphore Resource
wai_sem	Acquire Semaphore Resource
pol_sem	Acquire Semaphore Resource (Polling)
twai_sem	Acquire Semaphore Resource (with Timeout)

Eventflags

CRE_FLG	Create Eventflag (Static API)
set_flg / iset_flg	Set Eventflag
clr_flg	Clear Eventflag
wai_flg	Wait for Eventflag
pol_flg	Wait for Eventflag (Polling)
twai_flg	Wait for Eventflag (with Timeout)

Data queues

CRE_DTQ	Create Data Queue (Static API)
snd_dtq	Send to Data Queue
psnd_dtq / ipsnd_dtq	Send to Data Queue (Polling)
tsnd_dtq	Send to Data Queue (with Timeout)
fsnd_dtq / ifsnd_dtq	Forced Send to Data Queue
rcv_dtq	Receive from Data Queue
prcv_dtq	Receive from Data Queue (Polling)
trcv_dtq	Receive from Data Queue (with Timeout)

Mailboxes

CRE_MBX	Create Mailbox (Static API)
snd_mbx	Send to Mailbox
rcv_mbx	Receive from Mailbox
prcv_mbx	Receive from Mailbox (Polling)
trcv_mbx	Receive from Mailbox (with Timeout)

(6) Memory pool management functions

Fixed-sized memory pools

CRE_MPF	Create Fixed-Sized Memory Pool (Static API)
get_mpf	Acquire Fixed-Sized Memory Block
pget_mpf	Acquire Fixed-Sized Memory Block (Polling)
tget_mpf	Acquire Fixed-Sized Memory Block (with Timeout)
rel_mpf	Release Fixed-Sized Memory Block

(7) Time management functions

System time management

set_tim	Set System Time

get_tim	Reference System Time
isig_tim	Supply Time Tick

* If the kernel has a mechanism of updating the system time, isig_tim need not be supported.

Cyclic handlers

CRE_CYC	Create Cyclic Handler (Static API)
sta_cyc	Start Cyclic Handler Operation
stp_cyc	Stop Cyclic Handler Operation

(8) System state management functions

rot_rdq / irot_rdq	Rotate Task Precedence
get_tid / iget_tid	Reference Task ID in the RUNNING State
loc_cpu / iloc_cpu	Lock the CPU
unl_cpu / iunl_cpu	Unlock the CPU
dis_dsp	Disable Dispatching
ena_dsp	Enable Dispatching
sns_ctx	Reference Contexts
sns_loc	Reference CPU State
sns_dsp	Reference Dispatching State
sns_dpn	Reference Dispatch Pending State

(9) Interrupt management functions

DEF_INH	Define Interrupt Handler (Static API)

* If ATT_ISR is supported, DEF_INH need not be supported.

(11) System configuration management functions

DEF_EXC	Define CPU Exception Handler (Static API)
ATT_INI	Attach Initialization Routine (Static API)

7.4 Data Types

The data types, except those for packets, defined in the µITRON4.0 Specification are as follows:

B	Signed 8-bit integer
H	Signed 16-bit integer
W	Signed 32-bit integer
D	Signed 64-bit integer
UB	Unsigned 8-bit integer
UH	Unsigned 16-bit integer
UW	Unsigned 32-bit integer
UD	Unsigned 64-bit integer

VB	8-bit value with unknown data type
VH	16-bit value with unknown data type
VW	32-bit value with unknown data type
VD	64-bit value with unknown data type
VP	Pointer to an unknown data type
FP	Processing unit start address (pointer to a function)
INT	Signed integer for the processor
UINT	Unsigned integer for the processor
BOOL	Boolean value (TRUE or FALSE)
FN	Function code (signed integer)
ER	Error code (signed integer)
ID	Object ID number (signed integer)
ATR	Object attribute (unsigned integer)
STAT	Object state (unsigned integer)
MODE	Service call operational mode (unsigned integer)
PRI	Priority (signed integer)
SIZE	Memory area size (unsigned integer)
TMO	Timeout (signed integer, unit of time is implementation-defined)
RELTIM	Relative time (unsigned integer, unit of time is implementation-defined)
SYSTIM	System time (unsigned integer, unit of time is implementation-defined)
VP_INT	Pointer to an unknown data type, or a signed integer for the processor
ER_BOOL	Error code or a boolean value (signed integer)
ER_ID	Error code or an object ID number (signed integers and negative ID numbers cannot be represented)
ER_UINT	Error code or an unsigned integer (the number of available bits for an unsigned integer is one bit shorter than UINT)
TEXPTN	Bit pattern for the task exception code (unsigned integer)
FLGPTN	Bit pattern of the eventflag (unsigned integer)
T_MSG	Message header for a mailbox
T_MSG_PRI	Message header with a message priority for a mailbox
RDVPTN	Bit pattern of the rendezvous condition (unsigned integer)
RDVNO	Rendezvous number
OVRTIM	Processor time (unsigned integer, unit of time is implementation-defined)

INHNO	Interrupt handler number
INTNO	Interrupt number
I*XXXX*	Interrupt mask
EXCNO	CPU exception handler number

Among the above data types, the definition of the following data type is standardized:

```
typedef struct t_msg_pri {
        T_MSG     msgque ;   /* Message header */
        PRI       msgpri ;   /* Message priority */
} T_MSG_PRI ;
```

[Standard Profile]

The data types, except those for packets, that must be defined in the Standard Profile, their minimum number of bits, and their unit of time are as follows:

B	Signed 8-bit integer
H	Signed 16-bit integer
W	Signed 32-bit integer
UB	Unsigned 8-bit integer
UH	Unsigned 16-bit integer
UW	Unsigned 32-bit integer
VB	8-bit value with unknown data type
VH	16-bit value with unknown data type
VW	32-bit value with unknown data type
VP	Pointer to an unknown data type
FP	Processing unit start address (pointer to a function)
INT	Signed integer for the processor (16 or more bits)
UINT	Unsigned integer for the processor (16 or more bits)
BOOL	Boolean value (TRUE or FALSE)
FN	Function code (signed integer, 16 or more bits)
ER	Error code (signed integer, 8 or more bits)
ID	Object ID number (signed integer, 16 or more bits)
ATR	Object attribute (unsigned integer, 8 or more bits)
STAT	Object state (unsigned integer, 16 or more bits)
MODE	Service call operational mode (unsigned integer, 8 or more bits)
PRI	Priority (signed integer, 16 or more bits)
SIZE	Memory area size (unsigned integer, equal to the number of bits in a pointer)
TMO	Timeout (signed integer, 16 or more bits, unit of time is 1 msec)

RELTIM	Relative time (unsigned integer, 16 or more bits, unit of time is 1 msec)
SYSTIM	System time (unsigned integer, 16 or more bits, unit of time is 1 msec)
VP_INT	Pointer to an unknown data type, or a signed integer for the processor
ER_UINT	Error code or an unsigned integer (the number of available bits for an unsigned integer is one bit shorter than UINT)
TEXPTN	Bit pattern for the task exception code (unsigned integer, 16 or more bits)
FLGPTN	Bit pattern of the eventflag (unsigned integer, 16 or more bits)
T_MSG	Message header for a mailbox
T_MSG_PRI	Message header with a message priority for a mailbox
INHNO	Interrupt handler number (when DEF_INH is supported)
INTNO	Interrupt number (when ATT_ISR is supported)
EXCNO	CPU exception handler number

7.5 Packet Formats

(1) Task management functions

Task creation information packet:

```
typedef struct t_ctsk {
        ATR         tskatr ;     /* Task attribute */
        VP_INT      exinf ;      /* Task extended information */
        FP          task ;       /* Task start address */
        PRI         itskpri ;    /* Task initial priority */
        SIZE        stksz ;      /* Task stack size (in bytes) */
        VP          stk ;        /* Base address of task stack space */
        /* Other implementation specific fields may be added. */
} T_CTSK ;
```

Task state packet:

```
typedef struct t_rtsk {
        STAT        tskstat ;    /* Task state */
        PRI         tskpri ;     /* Task current priority */
        PRI         tskbpri ;    /* Task base priority */
        STAT        tskwait ;    /* Reason for waiting */
        ID          wobjid ;     /* Object ID number for which the task is
                                    waiting */
        TMO         lefttmo ;    /* Remaining time until timeout */
        UINT        actcnt ;     /* Activation request count */
        UINT        wupcnt ;     /* Wakeup request count */
        UINT        suscnt ;     /* Suspension count */
```

```
        /* Other implementation specific fields may be added. */
} T_RTSK ;
```

Task state packet (simplified version):

```
typedef struct t_rtst {
        STAT        tskstat ;     /* Task state */
        STAT        tskwait ;     /* Reason for waiting */
        /* Other implementation specific fields may be added. */
} T_RTST ;
```

(3) Task exception handling functions

Task exception handling routine definition information packet:

```
typedef struct t_dtex {
        ATR         texatr ;      /* Task exception handling routine
                                     attribute */
        FP          texrtn ;      /* Task exception handling routine start
                                     address */
        /* Other implementation specific fields may be added. */
} T_DTEX ;
```

Task exception handling state packet:

```
typedef struct t_rtex {
        STAT        texstat ;     /* Task exception state */
        TEXPTN      pndptn ;      /* Pending exception code */
        /* Other implementation specific fields may be added. */
} T_RTEX ;
```

(4) Synchronization and communication functions

Semaphore creation information packet:

```
typedef struct t_csem {
        ATR         sematr ;      /* Semaphore attribute */
        UINT        isemcnt ;     /* Initial semaphore resource count */
        UINT        maxsem ;      /* Maximum semaphore resource count */
        /* Other implementation specific fields may be added. */
} T_CSEM ;
```

Semaphore state packet:

```
typedef struct t_rsem {
        ID          wtskid ;      /* ID number of the task at the head of the
                                     semaphore's wait queue */
        UINT        semcnt ;      /* Current semaphore resource count */
        /* Other implementation specific fields may be added. */
} T_RSEM ;
```

Eventflag creation information packet:

```
typedef struct t_cflg {
        ATR         flgatr ;      /* Eventflag attribute */
        FLGPTN      iflgptn ;     /* Initial value of the eventflag bit
                                     pattern */
        /* Other implementation specific fields may be added. */
```

```
} T_CFLG ;
```

Eventflag state packet:

```
typedef struct t_rflg {
        ID          wtskid ;    /* ID number of the task at the head of the
                                   eventflag's wait queue */
        FLGPTN      flgptn ;    /* Current eventflag bit pattern */
        /* Other implementation specific fields may be added. */
} T_RFLG ;
```

Data queue creation information packet:

```
typedef struct t_cdtq {
        ATR         dtqatr ;    /* Data queue attribute */
        UINT        dtqcnt ;    /* Capacity of the data queue area (the
                                   number of data elements) */
        VP          dtq ;       /* Start address of the data queue area */
        /* Other implementation specific fields may be added. */
} T_CDTQ ;
```

Data queue state packet:

```
typedef struct t_rdtq {
        ID          stskid ;    /* ID number of the task at the head of the
                                   data queue's send-wait queue */
        ID          rtskid ;    /* ID number of the task at the head of the
                                   data queue's receive-wait queue */
        UINT        sdtqcnt ;   /* The number of data elements in the data
                                   queue */
        /* Other implementation specific fields may be added. */
} T_RDTQ ;
```

Mailbox creation information packet:

```
typedef struct t_cmbx {
        ATR         mbxatr ;    /* Mailbox attribute */
        PRI         maxmpri ;   /* Maximum message priority */
        VP          mprihd ;    /* Start address of the area for message
                                   queue headers for each message
                                   priority */
        /* Other implementation specific fields may be added. */
} T_CMBX ;
```

Mailbox state packet:

```
typedef struct t_rmbx {
        ID          wtskid ;    /* ID number of the task at the head of
                                   mailbox's wait queue */
        T_MSG *     pk_msg ;    /* Start address of the message packet at
                                   the head of the message queue */
        /* Other implementation specific fields may be added. */
} T_RMBX ;
```

(5) Extended synchronization and communication functions

Mutex creation information packet:

```
typedef struct t_cmtx {
        ATR         mtxatr ;      /* Mutex attribute */
        PRI         ceilpri ;     /* Mutex ceiling priority */
        /* Other implementation specific fields may be added. */
} T_CMTX ;
```

Mutex state packet:

```
typedef struct t_rmtx {
        ID          htskid ;      /* ID number of the task that locks the
                                     mutex */
        ID          wtskid ;      /* ID number of the task at the head of the
                                     mutex's wait queue */
        /* Other implementation specific fields may be added. */
} T_RMTX ;
```

Message buffer creation information packet:

```
typedef struct t_cmbf {
        ATR         mbfatr ;      /* Message buffer attribute */
        UINT        maxmsz ;      /* Maximum message size (in bytes) */
        SIZE        mbfsz ;       /* Size of message buffer area (in bytes) */
        VP          mbf ;         /* Start sddress of message buffer area */
        /* Other implementation specific fields may be added. */
} T_CMBF ;
```

Message buffer state packet:

```
typedef struct t_rmbf {
        ID          stskid ;      /* ID number of the task at the head of the
                                     message buffer's send-wait queue */
        ID          rtskid ;      /* ID number of the task at the head of the
                                     message buffer's receive-wait queue */
        UINT        smsgcnt ;     /* The number of messages in the message
                                     buffer */
        SIZE        fmbfsz ;      /* Size of free message buffer area in bytes,
                                     without the minimum control areas */
        /* Other implementation specific fields may be added. */
} T_RMBF ;
```

Rendezvous port creation information packet:

```
typedef struct t_cpor {
        ATR         poratr ;      /* Rendezvous port attribute */
        UINT        maxcmsz ;     /* Maximum calling message size (in
                                     bytes) */
        UINT        maxrmsz ;     /* Maximum return message size (in
                                     bytes) */
        /* Other implementation specific fields may be added. */
} T_CPOR ;
```

Rendezvous port state packet:

```
typedef struct t_rpor {
        ID          ctskid ;     /* ID number of the task at the head of the
                                    rendezvous port's call-wait queue */
        ID          atskid ;     /* ID number of the task at the head of the
                                    rendezvous port's accept-wait queue */
        /* Other implementation specific fields may be added. */
} T_RPOR ;
```

Rendezvous state packet:

```
typedef struct t_rrdv {
        ID          wtskid ;     /* ID number of the task in the termination
                                    waiting state for the rendezvous */
        /* Other implementation specific fields may be added. */
} T_RRDV ;
```

(6) Memory pool management functions

Fixed-sized memory pool creation information packet:

```
typedef struct t_cmpf {
        ATR         mpfatr ;     /* Fixed-sized memory pool attribute */
        UINT        blkcnt ;     /* Total number of memory blocks */
        UINT        blksz ;      /* Memory block size (in bytes) */
        VP          mpf ;        /* Start address of the fixed-sized memory
                                    pool area */
        /* Other implementation specific fields may be added. */
} T_CMPF ;
```

Fixed-sized memory pool state packet:

```
typedef struct t_rmpf {
        ID          wtskid ;     /* ID number of the task at the head of the
                                    fixed-sized memory pool's wait
                                    queue */
        UINT        fblkcnt ;    /* Number of free memory blocks in the
                                    fixed-sized memory pool */
        /* Other implementation specific fields may be added. */
} T_RMPF ;
```

Variable-sized memory pool creation information packet:

```
typedef struct t_cmpl {
        ATR         mplatr ;     /* Variable-sized memory pool attribute */
        SIZE        mplsz ;      /* Size of the variable-sized memory pool
                                    area (in bytes) */
        VP          mpl ;        /* Start address of the variable-sized
                                    memory pool area */
        /* Other implementation specific fields may be added. */
} T_CMPL ;
```

Variable-sized memory pool state packet:

```
typedef struct t_rmpl {
```

```
        ID          wtskid ;    /* ID number of the task at the head of the
                                   variable-sized memory pool's wait
                                   queue */
        SIZE        fmplsz ;    /* Total size of free memory blocks in the
                                   variable-sized memory pool (in
                                   bytes) */
        UINT        fblksz ;    /* Maximum memory block size available
                                   (in bytes) */
        /* Other implementation specific fields may be added. */
    } T_RMPL ;
```

(7) Time management functions

Cyclic handler creation information packet:

```
    typedef struct t_ccyc {
        ATR         cycatr ;    /* Cyclic handler attribute */
        VP_INT      exinf ;     /* Cyclic handler extended information */
        FP          cychdr ;    /* Cyclic handler start address */
        RELTIM      cyctim ;    /* Cyclic handler activation cycle */
        RELTIM      cycphs ;    /* Cyclic handler activation phase */
        /* Other implementation specific fields may be added. */
    } T_CCYC ;
```

Cyclic handler state packet:

```
    typedef struct t_rcyc {
        STAT        cycstat ;   /* Cyclic handler operational state */
        RELTIM      lefttim ;   /* Time left before the next activation */
        /* Other implementation specific fields may be added. */
    } T_RCYC ;
```

Alarm handler creation information packet:

```
    typedef struct t_calm {
        ATR         almatr ;    /* Alarm handler attribute */
        VP_INT      exinf ;     /* Alarm handler extended information */
        FP          almhdr ;    /* Alarm handler start address */
        /* Other implementation specific fields may be added. */
    } T_CALM ;
```

Alarm handler state packet:

```
    typedef struct t_ralm {
        STAT        almstat ;   /* Alarm handler operational state */
        RELTIM      lefttim ;   /* Time left before the activation */
        /* Other implementation specific fields may be added. */
    } T_RALM ;
```

Overrun handler definition information packet:

```
    typedef struct t_dovr {
        ATR         ovratr ;    /* Overrun handler attribute */
        FP          ovrhdr ;    /* Overrun handler start address */
        /* Other implementation specific fields may be added. */
    } T_DOVR ;
```

Overrun handler state packet:

```
typedef struct t_rovr {
        STAT        ovrstat ;     /* Overrun handler operational state */
        OVRTIM      leftotm ;     /* Remaining processor time */
        /* Other implementation specific fields may be added. */
} T_ROVR ;
```

(8) System state management functions

System state packet:

```
typedef struct t_rsys {
        /* Implementation specific fields */
} T_RSYS ;
```

(9) Interrupt management functions

Interrupt handler definition information packet:

```
typedef struct t_dinh {
        ATR         inhatr ;      /* Interrupt handler attribute */
        FP          inthdr ;      /* Interrupt handler start address */
        /* Other implementation specific fields may be added. */
} T_DINH ;
```

Interrupt service routine creation information packet:

```
typedef struct t_cisr {
        ATR         isratr ;      /* Interrupt service routine attribute */
        VP_INT      exinf ;       /* Interrupt service routine extended
                                     information */
        INTNO       intno ;       /* Interrupt number to which the interrupt
                                     service routine is to be attached */
        FP          isr ;         /* Interrupt service routine start sddress */
        /* Other implementation specific fields may be added. */
} T_CISR ;
```

Interrupt service routine state packet:

```
typedef struct t_risr {
        /* Implementaion-specific fields */
} T_RISR ;
```

(10) Service call management functions

Extended service call definition information packet:

```
typedef struct t_dsvc {
        ATR         svcatr ;      /* Extended service call attribute */
        FP          svcrtn ;      /* Extended service call routine start
                                     address */
        /* Other implementation specific fields may be added. */
} T_DSVC ;
```

(11) System configuration management functions

CPU exception handler definition information packet:

```
typedef struct t_dexc {
        ATR         excatr ;     /* CPU exception handler attribute */
        FP          exchdr ;     /* CPU exception handler start address */
        /* Other implementation specific fields may be added. */
} T_DEXC ;
```

Configuration information packet:

```
typedef struct t_rcfg {
        /* Implementation specific fields */
} T_RCFG ;
```

Version information packet:

```
typedef struct t_rver {
        UH          maker ;      /* Kernel maker's code */
        UH          prid ;       /* Identification number of the kernel */
        UH          spver ;      /* Version number of the ITRON
                                    Specification */
        UH          prver ;      /* Version number of the kernel */
        UH          prno[4] ;    /* Management information of the kernel
                                    product */
} T_RVER ;
```

7.6 Constants and Macros

(1) Object Attributes

TA_HLNG	0x00	Start a processing unit through a high-level language interface
TA_ASM	0x01	Start a processing unit through an assembly language interface
TA_TFIFO	0x00	Task wait queue is in FIFO order
TA_TPRI	0x01	Task wait queue is in task priority order
TA_MFIFO	0x00	Message queue is in FIFO order
TA_MPRI	0x02	Message queue is in message priority order
TA_ACT	0x02	Task is activated after the creation
TA_RSTR	0x04	Restricted task
TA_WSGL	0x00	Only one task is allowed to be in the waiting state for the eventflag
TA_WMUL	0x02	Multiple tasks are allowed to be in the waiting state for the eventflag
TA_CLR	0x04	Eventflag's bit pattern is cleared when a task is

		released from the waiting state for that eventflag
TA_INHERIT	0x02	Mutex uses the priority inheritance protocol
TA_CEILING	0x03	Mutex uses the priority ceiling protocol
TA_STA	0x02	Cyclic handler is in an operational state after the creation
TA_PHS	0x04	Cyclic handler is activated preserving the activation phase

(2) Service Call Operational Mode

TWF_ANDW	0x00	AND waiting condition for an eventflag
TWF_ORW	0x01	OR waiting condition for an eventflag

(3) Object States

TTS_RUN	0x01	RUNNING state
TTS_RDY	0x02	READY state
TTS_WAI	0x04	WAITING state
TTS_SUS	0x08	SUSPENDED state
TTS_WAS	0x0c	WAITING-SUSPENDED state
TTS_DMT	0x10	DORMANT state

TTW_SLP	0x0001	Sleeping state
TTW_DLY	0x0002	Delayed state
TTW_SEM	0x0004	Waiting state for a semaphore resource
TTW_FLG	0x0008	Waiting state for an eventflag
TTW_SDTQ	0x0010	Sending waiting state for a data queue
TTW_RDTQ	0x0020	Receiving waiting state for a data queue
TTW_MBX	0x0040	Receiving waiting state for a mailbox
TTW_MTX	0x0080	Waiting state for a mutex
TTW_SMBF	0x0100	Sending waiting state for a message buffer
TTW_RMBF	0x0200	Receiving waiting state for a message buffer
TTW_CAL	0x0400	Calling waiting state for a rendezvous
TTW_ACP	0x0800	Accepting waiting state for a rendezvous
TTW_RDV	0x1000	Terminating waiting state for a rendezvous
TTW_MPF	0x2000	Waiting state for a fixed-sized memory block
TTW_MPL	0x4000	Waiting state for a variable-sized memory block

TTEX_ENA	0x00	Task exception enabled state
TTEX_DIS	0x01	Task exception disabled state
TCYC_STP	0x00	Cyclic handler is in a non-operational state
TCYC_STA	0x01	Cyclic handler is in an operational state
TALM_STP	0x00	Alarm handler is in a non-operational state

TALM_STA	0x01	Alarm handler is in an operational state
TOVR_STP	0x00	Processor time limit is not set
TOVR_STA	0x01	Processor time limit is set

(4) Other constants

TSK_SELF	0	Specifying invoking task
TSK_NONE	0	No applicable task
TPRI_SELF	0	Specifying the base priority of the invoking task
TPRI_INI	0	Specifying the initial priority of the task

(5) Macros

ER mercd = MERCD (ER ercd)

This macro retrieves the main error code from an error code.

ER sercd = SERCD (ER ercd)

This macro retrieves the sub error code from an error code.

7.7 Kernel Configuration Constants and Macros

(1) Priority Range

TMIN_TPRI	Minimum task priority (= 1)
TMAX_TPRI	Maximum task priority
TMIN_MPRI	Minimum message priority (= 1)
TMAX_MPRI	Maximum message priority

(2) Version Information

TKERNEL_MAKER	Kernel maker code
TKERNEL_PRID	Identification number of the kernel
TKERNEL_SPVER	Version number of the ITRON Specification
TKERNEL_PRVER	Version number of the kernel

(3) Maximum Nesting/Queueing Count

TMAX_ACTCNT	Maximum activation request count
TMAX_WUPCNT	Maximum wakeup request count
TMAX_SUSCNT	Maximum suspension count

(4) Number of Bits in Bitpatterns

TBIT_TEXPTN	Number of bits in the task exception code

TBIT_FLGPTN	Number of bits in an eventflag
TBIT_RDVPTN	Number of bits in a rendezvous condition

(5) Time Tick Period

TIC_NUME	Time tick period numerator
TIC_DENO	Time tick period denominator

(6) Required Memory Size

SIZE dtqsz = TSZ_DTQ (UINT dtqcnt)

Total required size of the data queue area in bytes necessary to store dtqcnt data elements

SIZE mprihdsz = TSZ_MPRIHD (PRI maxmpri)

Total required size in bytes of the area for message queue headers for each message priority, when the maximum message priority is maxmpri

SIZE mbfsz = TSZ_MBF (UINT msgcnt, UINT msgsz)

Approximate required size of the message buffer area in bytes necessary to store msgcnt messages each consisting of msgsz bytes

SIZE mpfsz = TSZ_MPF (UINT blkcnt, UINT blksz)

Total required size of the fixed-size memory pool area in bytes necessary to allocate blkcnt memory blocks each of size blksz bytes

SIZE mplsz = TSZ_MPL (UINT blkcnt, UINT blksz)

Approximate size in bytes necessary to allocate blkcnt memory blocks each of size blksz bytes

(7) Others

TMAX_MAXSEM	Maximum value of the maximum definable semaphore resource count

7.8 Error Code List

E_SYS	–5	System error
E_NOSPT	–9	Unsupported function
E_RSFN	–10	Reserved function code
E_RSATR	–11	Reserved attribute
E_PAR	–17	Parameter error
E_ID	–18	Invalid ID number
E_CTX	–25	Context error

E_MACV	–26	Memory access violation
E_OACV	–27	Object access violation
E_ILUSE	–28	Illegal service call use
E_NOMEM	–33	Insufficient memory
E_NOID	–34	No ID number available
E_OBJ	–41	Object state error
E_NOEXS	–42	Non-existent object
E_QOVR	–43	Queue overflow
E_RLWAI	–49	Forced release from waiting
E_TMOUT	–50	Polling failure or timeout
E_DLT	–51	Waiting object deleted
E_CLS	–52	Waiting object state changed
E_WBLK	–57	Non-blocking call accepted
E_BOVR	–58	Buffer overflow

7.9 Function Code List

	–0	–1	–2	–3
–0x01	reserved	reserved	reserved	reserved
–0x05	cre_tsk	del_tsk	act_tsk	can_act
–0x09	sta_tsk	ext_tsk	exd_tsk	ter_tsk
–0x0d	chg_pri	get_pri	ref_tsk	ref_tst
–0x11	slp_tsk	tslp_tsk	wup_tsk	can_wup
–0x15	rel_wai	sus_tsk	rsm_tsk	frsm_tsk
–0x19	dly_tsk	reserved	def_tex	ras_tex
–0x1d	dis_tex	ena_tex	sns_tex	ref_tex
–0x21	cre_sem	del_sem	sig_sem	reserved
–0x25	wai_sem	pol_sem	twai_sem	ref_sem
–0x29	cre_flg	del_flg	set_flg	clr_flg
–0x2d	wai_flg	pol_flg	twai_flg	ref_flg
–0x31	cre_dtq	del_dtq	reserved	reserved
–0x35	snd_dtq	psnd_dtq	tsnd_dtq	fsnd_dtq
–0x39	rcv_dtq	prcv_dtq	trcv_dtq	ref_dtq
–0x3d	cre_mbx	del_mbx	snd_mbx	reserved
–0x41	rcv_mbx	prcv_mbx	trcv_mbx	ref_mbx
–0x45	cre_mpf	del_mpf	rel_mpf	reserved
–0x49	get_mpf	pget_mpf	tget_mpf	ref_mpf
–0x4d	set_tim	get_tim	cre_cyc	del_cyc
–0x51	sta_cyc	stp_cyc	ref_cyc	reserved

–0x55	rot_rdq	get_tid	reserved	reserved
–0x59	loc_cpu	unl_cpu	dis_dsp	ena_dsp
–0x5d	sns_ctx	sns_loc	sns_dsp	sns_dpn
–0x61	ref_sys	reserved	reserved	reserved
–0x65	def_inh	cre_isr	del_isr	ref_isr
–0x69	dis_int	ena_int	chg_*ixx*	get_*ixx*
–0x6d	def_svc	def_exc	ref_cfg	ref_ver
–0x71	iact_tsk	iwup_tsk	irel_wai	iras_tex
–0x75	isig_sem	iset_flg	ipsnd_dtq	ifsnd_dtq
–0x79	irot_rdq	iget_tid	iloc_cpu	iunl_cpu
–0x7d	isig_tim	reserved	reserved	reserved
–0x81	cre_mtx	del_mtx	unl_mtx	reserved
–0x85	loc_mtx	ploc_mtx	tloc_mtx	ref_mtx
–0x89	cre_mbf	del_mbf	reserved	reserved
–0x8d	snd_mbf	psnd_mbf	tsnd_mbf	reserved
–0x91	rcv_mbf	prcv_mbf	trcv_mbf	ref_mbf
–0x95	cre_por	del_por	cal_por	tcal_por
–0x99	acp_por	pacp_por	tacp_por	fwd_por
–0x9d	rpl_rdv	ref_por	ref_rdv	reserved
–0xa1	cre_mpl	del_mpl	rel_mpl	reserved
–0xa5	get_mpl	pget_mpl	tget_mpl	ref_mpl
–0xa9	cre_alm	del_alm	sta_alm	stp_alm
–0xad	ref_alm	reserved	reserved	reserved
–0xb1	def_ovr	sta_ovr	stp_ovr	ref_ovr
–0xb5	reserved	reserved	reserved	reserved
–0xb9	reserved	reserved	reserved	reserved
–0xbd	reserved	reserved	reserved	reserved
–0xc1	acre_tsk	acre_sem	acre_flg	acre_dtq
–0xc5	acre_mbx	acre_mtx	acre_mbf	acre_por
–0xc9	acre_mpf	acre_mpl	acre_cyc	acre_alm
–0xcd	acre_isr	reserved	reserved	reserved
–0xd1	reserved	reserved	reserved	reserved
–0xd5	reserved	reserved	reserved	reserved
–0xd9	reserved	reserved	reserved	reserved
–0xdd	reserved	reserved	reserved	reserved
–0xe1	implementation-specific service calls			
–0xe5	implementation-specific service calls			
–0xe9	implementation-specific service calls			
–0xed	implementation-specific service calls			

–0xf1	implementation-specific service calls
–0xf5	implementation-specific service calls
–0xf9	implementation-specific service calls
–0xfd	implementation-specific service calls

Index

This is an index of the terms used in the main body of the µITRON4.0 Specification (Chapter 2 to Chapter 5). The number refers to the page where the term is defined or explained.

A

B

C

D