ITRON TCP/IP API Specification

Ver. 2.00.00

Supervised by Ken Sakamura
Edited and Published by
TRON ASSOCIATION

ITRON TCP/IP API Specification (Ver. 2.00.00)

- § TRON is the abbreviation of "The Real-time Operating system Nucleus."
- § ITRON is the abbreviation of "Industrial TRON."
- § TRON, ITRON, and T-Kernel do not refer to any specific product or products.

Preface

Networks constitute absolutely indispensable infrastructure for realizing ubiquitous computing environments.

Enjoyment of information services by anyone, anytime, and anywhere requires that various devices be interconnected through networks, constantly exchanging the most up-to-date information, and functioning in harmony.

TCP/IP is the major network protocol for such ubiquitous computing environments. Thus far, a socket interface has been the typical API for TCP/IP. Since it was originally developed for servers and personal computers, however, it has been unable to fully satisfy the requirements of systems that have strict limitations on hardware resources, such as embedded systems. The TRON project thus designed the ITRON TCP/IP API Specification as an API specification that is optimal for embedded use. Version 1.0 was created in 1998. In the eight years since then, this specification has established a place as a representative API for the use of embedded systems on networks and been incorporated in many products.

Networking and embedded-system technologies have changed greatly compared with those available eight years ago. For example, the household Internet penetration rate in Japan was then about 10%, but is now over 85%. Most remarkable is the growth in the number of users accessing the Internet through embedded systems such as mobile phones or PDAs; in statistics at the end of 2005, the number of users going through embedded systems to access the Internet had surpassed the number going through personal computers. Thus, the demand has come to be for embedded systems with complexity and advanced functionality, including functionality for network connection, rather than the conventional simple ones. In response to this demand for advanced embedded systems, the TRON Project deployed μITRON4.0 and then the T-Kernel as real-time kernel specifications. TCP/IP which forms the base of the Internet has also greatly changed. The IPv6 specification was still under development when we created version 1.0 of the ITRON TCP/IP API Specification; now, the specification has been settled and is being applied for actual operation.

These changes in the technological situations for networks and embedded systems have made it necessary to adapt the ITRON TCP/IP API Specification in response to new technologies. We have thus revised the ITRON TCP/IP API Specification to produce Version 2.0 with a focus on

support for the T-Kernel, and μITRON4.0, and the IPv6 specification.

To enable the reuse of software based on the earlier version, we have also tried to maintain maximum compatibility with the earlier specification while revising it to support new technologies. We have also carefully considered facilitating the porting of software for earlier versions to the new real-time OS with reference to this new specification. We certainly want to see existing software developed under Version 1.0 of the ITRON TCP/IP API Specification ported to run under current operating systems such as the T-Kernel.

Networking technology will be indispensable in developing embedded systems of the future, and it is not hard to envisage almost all embedded systems incorporating networking functions.

I will be pleased if this specification is able to meet the needs of many embedded-system engineers in the future.

July, 2006
Ken Sakamura
Project Leader, TRON Project

Contents

Chapter 1 Introduction

1.1 Background and Scope of Standardization

The trend towards growing scale and complexity for functions incorporated in the embedded systems of recent years has been creating an increasingly broad consciousness of the importance of software components for embedded systems due to the necessity of improving the basic conditions of application software development. As software components are used in an ever-wider range of applications, standardization of the components has become a key issue. Against this background, TRON Association has been working on standardization of the interfaces for software components, with a focus on selected fields that have the strongest needs in relation to the ITRON-specification OS.

One of the most important software components for an embedded system is a TCP/IP protocol stack. The socket interface is currently in widespread use as the TCP/IP application program interface (API). However, its disadvantages in terms of application in embedded systems (and particularly in small-scale embedded systems) have created the need to standardize a TCP/IP API that is more suitable for embedded systems.

To satisfy this need, in 1997 TRON Association formed a specialized group to study the standardization of a TCP/IP API suitable for embedded systems. This was based on a proposal by the ITRON Technical Committee of that time (today's ITRON Specification Study Group). The group continued the study from March 1997 to April 1998 and determined Version 1.0 of the TCP/IP API Specification, which was approved as an ITRON specification. Version 1.0 is a standard specification of a C-language API for TCP and UDP protocols on IPv4. In keeping with the basic policy of the TRON project, this specification is open to anyone and carries no license fee.

When we were drawing up Version 1.0, we had the μITRON3.0 specifications, after which we went to μITRON4.0 and then also deployed the T-Kernel. As the IPv6 specification had not been completed at the time, we left the application of IPv6 for later consideration. In the five years since its development, Version 1.0 has been widely recognized as a TCP/IP API that is suitable for embedded systems, and many products that conform to this specification have appeared. The permeation of this specification has been accompanied by requests from users for the support for the latest ITRON-specification OS and IPv6. We have thus reviewed the TCPI/IP API Specification and implemented Version 2.0 to support the T-Kernel, μITRON4.0, and IPv6 specifications while maintaining maximum compatibility with Version 1.0.

In addition to the API functions prescribed in this specification, the TCP/IP protocol stack in practical use requires further API functions for interface control (IP address,

netmask, and broadcast address setting and reference), routing table setting and reference, and ARP table setting and reference, but this version of the specification does not prescribe a standard for such functions. Handling of ICMP is not prescribed in this specification and is left implementation-dependent.

1.2 Requirements for Embedded Systems

Embedded systems have the following general characteristics.

(1) Hardware resources are strictly limited (i.e., processor capabilities and memory capacity). The program and data must thus be as small as possible to draw full performance from the processor.

(2) Some kind of real-time capability will be required. Fine control of the protocol stack is also required.

(3) High reliability.

Specifically, a protocol stack for embedded systems should have the following features:

(a) minimization of the memory area for buffers and the number of times data are copied;

(b) no dynamic memory management done within the protocol stack is desirable;

(c) support for an asynchronous interface or non-blocking calls; and

(d) comprehensibility of detailed information on errors for the respective API functions is desirable.

The socket interface does not satisfy all of these requirements. For example, the socket management block and the memory area for queuing the received UDP packets have to be managed dynamically. In addition, the maximum size of the latter area is not specifiable on the basis of the protocol, so that situations where memory becomes insufficient for the protocol stack area are not avoidable. Such a problem (memory becoming insufficient in regions that are not visible from the application program) is not acceptable for embedded systems, which must always operate in the expected manner.

The socket interface has another disadvantage in that it was designed to match the UNIX process model and thus does not work well with the task model supported by a real-time OS such as the ITRON-specification OS.

Note: UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/OPEN Company Limited.

1.3 API Design Policies

The goal of the TCP/IP API defined in this specification is to implement the TCP/IP protocol stack that satisfies the requirements described above. In addition, we had the following design policies.

- The specification should be based on the socket interface, since this is familiar to many programmers and has been used in developing many software products. The design of the API thus had to be such that an API compatible with the socket interface is realizable as a library on top of it.
- Consideration should be given to allowing some small-scale embedded systems to require only static settings to be made at the time of system configuration.
- Although usage on top of an ITRON-specification OS should be the prime assumption and the API should be prescribed in line with the conventions of the ITRON specification, applicability to other real-time OSs should also be considered.

1.4 General Concepts

1.4.1 API Levels and Static APIs

- APIs and their functions are classified into standard functions and extended functions according to the level of necessity. The TCP/IP protocol stacks that will be available on the market should at least implement the standard functions.
- Each object-creating API provides a static method (called a static API) for specifying object creation information in the system configuration file; the object specified in the system configuration file is automatically created when the protocol stack is initialized. The names of static APIs are written in upper case letters and thus they are distinguished from standard APIs (called dynamic APIs).

1.4.2 API Return Values and Error Codes

- According to the conventions of the ITRON and T-Kernel specifications (called the kernel specifications), each API returns a negative-value error code if an error occurs, or 0 or a positive value if it completes execution normally. The meaning of the return value for normal completion is defined for each API.
- An error code consists of a main error code and a sub-error code. Both the main error code and sub-error code are negative, and the resulting combined error code is also negative.
- The mnemonics, meanings, and values of the main error codes are standardized to be the same as those of the error codes defined in the kernel specification. The error codes that are required for TCP/IP API but not defined in the kernel specification (E_WBLK, E_CLS, and E_BOVR in the μITRON3.0 Specification and T-Kernel Specification) are defined additionally. The values of the error codes are defined in

eight bits (negative values from -128 to -1) in accordance with the kernel specification.

- Sub-error codes are implementation-dependent. This specification defines only the main error codes as the error codes to be returned by each API. To clarify an error cause, sub-error codes should be used.
- The following error codes can be returned by all APIs, thus they are not shown in the description of each API in this specification. It is implementation-dependent which particular APIs actually return which error codes.

E_SYS	System error (internal error in the protocol stack)
E_NOMEM	Insufficient memory
E_NOSPT	Unsupported function
E_MACV	Memory access violation

1.4.3 Communication End Point

- A communication end point for each protocol, which is identified by an IP address and a port number, is considered to be equivalent to an object in the ITRON kernel specification. Communication end points are classified into several types according to the protocol and function. In this aspect, communication end points are at a lower level of abstraction than sockets although communication end points are the equivalent of sockets in the socket interface.
- Communication end points are identified by ID numbers uniquely assigned within each communication end point type throughout the entire system.
- This specification does not adopt abstraction equivalent to the file descriptor feature in the socket interface; APIs directly manipulate communication end points in this specification. This difference is most noticeable in the TCP disconnection procedure. In the socket interface, the close function returns as soon as data transmission ends without waiting for the end of the disconnection procedure, to cancel the association between the file descriptor and a socket, and the file descriptor can be used for another purpose immediately after the close function has returned. In contrast, the tcp_cls_cep execution (without timeout setting) in this specification waits until the communication end point becomes ready for reuse.
- This specification assumes that TCP communication end point transmit/receive buffers are window buffers. Whether transmit/receive buffers are prepared by the application or by the protocol stack depends on the implementation. Refer to the description of the tcp_cre_cep function for creating a TCP communication end point.

1.4.4 Timeout and Non-Blocking Calls

- Timeout and non-blocking call features are made available to APIs that might enter the waiting state.
- When an API's process is not completed within a specified time, the timeout feature cancels any further processing and the API immediately returns. In this case, the

API returns an E_TMOUT error. The object state remains unchanged upon returning from the timed-out API. However, there are some exceptional cases where an API due to its nature might prevent the object from proper restoration after cancellation of the processing.

- In the non-blocking call feature, an API that should enter the waiting state immediately returns with an E_WBLK error but the processing continues. The application program is reported through a callback (to be described later) when the process is completed or when it is canceled. When an API with the non-blocking feature is called to pass a data area pointer, the area should not be used for another purpose until the process is completed because the processing continues even after the API has returned.
- The timeout processing with the timeout duration set to 0 is referred to as "polling". The polling feature differs from the non-blocking call feature in that polling cancels the processing while non-blocking continues the processing.
- Processing of an API is referred to as "pending" when it is in the waiting state within the API or when it continues operation due to a non-blocking call.
- The functional descriptions of the APIs in this specification contain the behavior when the APIs have no timeout setting; that is, the behavior when the APIs wait forever. When an API is called with a timeout duration set, the description "entering (moved to) the waiting state" in the functional descriptions of the APIs do not imply waiting forever; the waiting state is canceled and the API returns with E_TMOUT as the return value when the duration expires. When an API is called with the non-blocking feature, the API does not enter the waiting state but immediately returns with E_TMOUT as the return value.
- The timeout should be specified in accordance with the ITRON kernel specification; a positive value specifies the length of the timeout duration (millisecond is recommended as the unit of setting), TMO_POL specifies polling, and TMO_FEVR specifies that the timeout duration should be forever. TMO_NBLK specifies the non-blocking call feature.

1.4.5 Callbacks

- Callbacks are used to report the application program that an event has occurred in the protocol stack. Callback events are broadly classified into non-blocking call completion report and other events.
- Callback routines are defined by the application program and executed only in task contexts.
- A single callback routine is defined for each communication end point, except for TCP reception points, which do not have callback routines. The type of the event that has generated a callback is passed to the callback routine as an argument.
- The callback routine for one communication end point is never nested; that is, for a communication end point, no other callback routine is called until the current callback routine returns.

[Rationale]

Although the callbacks of this specification have a disadvantage in that they provide a low level of safety with respect to the software architecture, they offer high run-time efficiency, which should have a priority in small-scale embedded systems that are the main target of the ITRON-specification OS.

1.4.6 Relationships between APIs and Tasks

- Each API of this specification always works in the same manner given the same parameters, regardless of the task that calls it; that is, the API of this specification does not assign any resource to tasks. This means that there is no resource that the protocol stack should release when a task processing is completed.
- When rel_wai is issued to a task that has entered the waiting state in execution of an API of this specification, the API returns an E_RLWAI error. When ter_tsk is issued in the same situation, the behavior is implementation-dependent.

1.5 General Definitions

1.5.1 Data Structures and Data Types

(1) Data Structure for Containing an IP Address and a Port Number

```
typedef struct t_ipv4ep {
    UW      ipaddr;         /* IP address */
    UH      portno;         /* Port number */
} T_IPV4EP;
```

[Supplemental Information]

An IP address should be assigned to the ipaddr field in the network order (big endian). A value can be directly assigned without using memory copy functions such as a "bcopy".

(2) Data Structures for Creating Objects

```
typedef struct t_tcp_crep {
    ATR         repatr;     /* TCP reception point attribute */
    T_IPV4EP    myaddr;     /* Local IP address and port number */
    (Other implementation-dependent fields may be added)
} T_TCP_CREP;

typedef struct t_tcp_ccep {
    ATR         cepatr;     /* TCP communication end point attribute */
    VP          sbuf;       /* Start address of transmit window buffer */
    INT         sbufsz;     /* Size of transmit window buffer */
    VP          rbuf;       /* Start address of receive window buffer */
    INT         rbufsz;     /* Size of receive window buffer */
    FP          callback;   /* Callback routine */
```

```
    (Other implementation-dependent fields may be added)
} T_TCP_CCEP;

typedef struct t_udp_ccep {
    ATR         cepatr;     /* UDP communication end point attribute */
    T_IPV4EP    myaddr;     /* Local IP address and port number */
    FP          callback;   /* Callback routine */
    (Other implementation-dependent fields may be added)
} T_UDP_CCEP;
```

[Supplemental Information]

A desired operation can be precisely specified for each object through the attributes of the communication end point; for example, the initial value can be specified for each communication end point.

1.5.2 Utility Functions and Macros

(1) Byte Order Converting Functions and Macros

UW nl = htonl(UW hl)	Converts a 32-bit value arranged in the host byte order into one arranged in the network byte order.
UH ns = htons(UH hs)	Converts a 16-bit value arranged in the host byte order into one arranged in the network byte order.
UW hl = ntohl(UW nl)	Converts a 32-bit value arranged in the network byte order into one arranged in the host byte order.
UH hs = ntohs(UH ns)	Converts a 16-bit value arranged in the network byte order into one arranged in the host byte order.

(2) Error Code Retrieving Functions and Macros

ER mercd = mainercd(ER ercd)	Retrieves the main error code from an error code.
ER sercd = subercd(ER ercd)	Retrieves the sub-error code from an error code.

1.5.3 Constants

(1) General Constant

NADR	Invalid address (See Appendix for details)

(2) API Function Codes and Event Codes

TFN_TCP_CRE_REP	–0x201 (0xfdff)
TFN_TCP_DEL_REP	–0x202 (0xfdfe)
TFN_TCP_CRE_CEP	–0x203 (0xfdfd)
TFN_TCP_DEL_CEP	–0x204 (0xfdfc)
TFN_TCP_ACP_CEP	–0x205 (0xfdfb)
TFN_TCP_CON_CEP	–0x206 (0xfdfa)
TFN_TCP_SHT_CEP	–0x207 (0xfdf9)
TFN_TCP_CLS_CEP	–0x208 (0xfdf8)
TFN_TCP_SND_DAT	–0x209 (0xfdf7)
TFN_TCP_RCV_DAT	–0x20a (0xfdf6)
TFN_TCP_GET_BUF	–0x20b (0xfdf5)
TFN_TCP_SND_BUF	–0x20c (0xfdf4)
TFN_TCP_RCV_BUF	–0x20d (0xfdf3)
TFN_TCP_REL_BUF	–0x20e (0xfdf2)
TFN_TCP_SND_OOB	–0x20f (0xfdf1)
TFN_TCP_RCV_OOB	–0x210 (0xfdf0)
TFN_TCP_CAN_CEP	–0x211 (0xfdef)
TFN_TCP_SET_OPT	–0x212 (0xfdee)
TFN_TCP_GET_OPT	–0x213 (0xfded)
TFN_TCP_ALL	0

TEV_TCP_RCV_OOB	0x201	TCP urgent data reception

(Other events should be defined by the implementation)

TFN_UDP_CRE_CEP	–0x221 (0xfddf)
TFN_UDP_DEL_CEP	–0x222 (0xfdde)
TFN_UDP_SND_DAT	–0x223 (0xfddd)
TFN_UDP_RCV_DAT	–0x224 (0xfddc)
TFN_UDP_CAN_CEP	–0x225 (0xfddb)
TFN_UDP_SET_OPT	–0x226 (0xfdda)
TFN_UDP_GET_OPT	–0x227 (0xfdd9)
TFN_UDP_ALL	0

TEV_UDP_RCV_DAT	0x221	UDP packet reception

(Other events should be defined by the implementation)

(3) Main Error Codes (see Appendix for details)

E_OK	Normal completion
E_SYS	System error
E_NOMEM	Insufficient memory
E_NOSPT	Unsupported function

E_RSATR	Reserved attribute
E_PAR	Parameter error
E_ID	Invalid ID number
E_NOEXS	Non-existent object
E_OBJ	Object state error
E_MACV	Memory access violation
E_QOVR	Queuing overflow
E_DLT	Waiting object deleted
E_WBLK	Non-blocking call accepted
E_TMOUT	Polling failure or timeout
E_RLWAI	Forced release from waiting
E_CLS	Disconnected
E_BOVR	Buffer overflow

[Supplemental Information]

The error codes other than E_OK and their values are defined in each kernel specification. Note that the error codes not defined in the kernel specification (E_WBLK, E_CLS, and E_BOVR) should be additionally defined.

E_WBLK indicates that a non-blocking call has been accepted and has returned but the processing continues. The completion or cancellation of the processing is reported through a callback.

E_CLS indicates that the connection has been abnormally disconnected. In some cases, an abnormal disconnection might occur while waiting for completion of a processing such as transmission or reception. If necessary, the waiting state should be canceled in such a case.

E_BOVR indicates that there is no sufficient area to hold the received data. The amount of received data that fits into the available area has been stored and the remaining data has been discarded. Unlike other errors, if this error code is returned, the API processing has been done and the object state has been changed.

[Rationale]

The error code for abnormal disconnection has been added because this error involves release from the waiting state so E_OBJ is not suitable. Although a possible alternative specification is to return an E_OBJ error if preceded the function call and to return an E_CLS error if disconnection followed entry to the waiting state, handling of this method by applications was considered difficult.

(4) Timeout Specification (see Appendix for details)

TMO_POL	Polling
TMO_FEVR	Waiting forever

TMO_NBLK	Non-blocking call

(5) Special IP Address and Port Numbers

IPV4_ADDRANY	0	IP address specification omitted
TCP_PORTANY	0	TCP port number specification omitted
UDP_PORTANY	0	UDP port number specification omitted

Chapter 2 APIs for TCP

2.1 Overview

- Two types of end points are available for use with TCP; TCP reception points (abbreviated to rep) for waiting for connection requests from remote hosts and TCP communication end points (abbreviated to cep) to be used as the end points for connection.
- A TCP communication end point makes transitions among eight states in the API specification: the "non-existent", "unused", "passive-open-waiting", "active-open-waiting", "connected", "transmission-end", "disconnected", and "closing" states (see figure 1). The seven states other than the non-existent state are collectively called "existent", the six states other than the non-existent and unused states are called "in-use", and the five states other than the connected, transmission-end, and disconnected states are called "unconnected".
- In addition to the APIs equivalent to read and write in the socket interface, efficient APIs that can reduce the data copying count should be available. These are called copy-saving APIs.

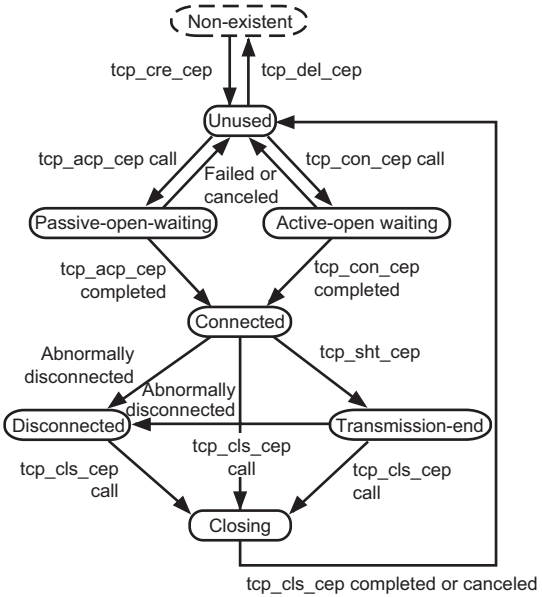

Figure 1. State Transition of TCP Communication End Point

[Rationale]

The copy-saving APIs available in this specification allow the user to directly access the buffers managed by the protocol stack. Another more efficient method is to allow the

protocol stack to directly use the buffers managed by the application. Although this alternative method is effective, the application program becomes complex and efficiency is not always improved when the data length is short. Accordingly, this method is not adopted in this specification.

The standard APIs can be implemented by using the copy-saving APIs as shown in Appendix C.1. In this specification, however, both types of API are prepared because such copy-saving API use is not always efficient depending on the method of protocol-stack implementation.

2.2 Creating and Deleting TCP Reception Points

TCP_CRE_REP Create TCP Reception Point
Static API...[Standard Function]
tcp_cre_rep Dynamic API...[Extended Function]

[Static API]

TCP_CRE_REP(ID repid, { ATR repatr, { UP myipaddr, UH myportno }});

[C Language API]

ER ercd = tcp_cre_rep(ID repid, T_TCP_CREP *pk_crep);

[Parameter]

ID	repid	TCP reception point ID
T_TCP_CREP	*pk_crep	TCP reception point creation information

pk_crep includes

ATR	repatr	TCP reception point attribute
T_IPV4EP	myaddr	Local IP address (myipaddr) and port number (myportno)

(Other implementation-dependent parameters may be added)

[Return Parameter]

ER	ercd	Error code

[Error Code]

E_OK	Normal completion
E_ID	Invalid ID number
E_RSATR	Reserved attribute
E_PAR	Parameter error (pk_crep address, IP address, or port number is invalid)
E_OBJ	Object state error (a TCP reception point is already created for the specified ID, the specified port number is already in use, or there is a privileged port violation)

[Functional Description]

These APIs create a TCP reception point with the specified ID and move the reception point to the waiting state for connection with the specified IP address and port number. When IPV4_ADDRANY is specified as the local IP address, the reception point waits for connection requests to all IP addresses assigned to the local host. When a particular IP address is specified, the reception point waits for only the connection request to the specified IP address.

Usage of the TCP reception point attribute and details of privileged ports are implementation-dependent.

[Rationale]

This specification does not provide a standard method for specifying an individual maximum count of connection requests to be queued for each TCP reception point. Queuing connection requests is important, but simply specifying one maximum count to be applied in common to all reception points will be sufficient for the purpose, and specifying an individual maximum count for each reception point is left implementation-dependent.

tcp_del_rep Delete TCP Reception Point

Dynamic API...[Extended Function]

[C Language API]

```
ER ercd = tcp_del_rep(ID repid);
```

[Parameter]

ID	repid	TCP reception point ID

[Return Parameter]

ER	ercd	Error code

[Error Code]

E_OK	Normal completion
E_ID	Invalid ID number
E_NOEXS	Non-existent object

[Functional Description]

This API deletes the specified TCP reception point and cancels the waiting state for connection. An E_DLT error will be returned to the tcp_acp_cep that is waiting for a connection request at the deleted TCP reception point.

2.3 Creating and Deleting TCP Communication End Points

tcp_CRE_CEP Create TCP Communication End Point
Static API...[Standard Function]
tcp_cre_cep Dynamic API...[Extended Function]

[Static API]

TCP_CRE_CEP(ID cepid, { ATR cepatr, VP sbuf, INT sbufsz, VP rbuf, INT rbufsz, FP callback });

[C Language API]

ER ercd = tcp_cre_cep(ID cepid, T_TCP_CCEP *pk_ccep);

[Parameter]

ID	cepid	TCP communication end point ID
T_TCP_CCEP	*pk_ccep	TCP communication end point creation information

pk_ccep includes

ATR	cepatr	TCP communication end point attribute
VP	sbuf	Start address of the transmit window buffer
INT	sbufsz	Size of the transmit window buffer
VP	rbuf	Start address of the receive window buffer
INT	rbufsz	Size of the receive window buffer
FP	callback	Callback routine address. Specify NULL when not using a callback routine.

(Other implementation-dependent parameters may be added)

[Return Parameter]

ER	ercd	Error code

[Error Code]

E_OK	Normal completion
E_ID	Invalid ID number
E_RSATR	Reserved attribute
E_PAR	Parameter error (pk_ccep address, sbuf, sbufsz, rbuf, rbufsz, or callback is invalid)
E_OBJ	Object state error (a TCP communication end point is already created for the specified ID)

[Functional Description]

These APIs create a TCP communication end point with the specified ID. The created TCP communication end point can be used both for a passive open and an active open.

In the implementation in which the protocol stack allocates window buffers, NADR

should be specified as the start address of window buffers. Setting window buffer sizes is valid even in this case. Another possible implementation is to allocate a window buffer in the protocol stack only when NADR is specified and otherwise to use a given buffer. Usage of the TCP communication end point attribute is implementation-dependent.

---

tcp_del_cep Delete TCP Communication End Point

Dynamic API...[Extended Function]

[C Language API]

ER ercd = tcp_del_cep(ID cepid);

[Parameter]

ID	cepid	TCP communication end point ID

[Return Parameter]

ER	ercd	Error code

[Error Code]

E_OK	Normal completion
E_ID	Invalid ID number
E_NOEXS	Non-existent object
E_OBJ	Object state error (the specified TCP communication end point is already in use)

[Functional Description]

This API deletes the specified TCP communication end point. If the TCP communication end point specified for deletion is already in use, an E_OBJ error will be returned.

2.4 Connecting and Disconnecting

---

tcp_acp_cep Wait for Connection Request (Passive Open)

[Standard Function]

[C Language API]

ER ercd = tcp_acp_cep(ID cepid, ID repid, T_IPV4EP *p_dstaddr, TMO tmout);

[Parameter]

ID	cepid	TCP communication end point ID
ID	repid	TCP reception point
TMO	tmout	Timeout

[Return Parameter]

T_IPV4EP	dstaddr	Remote IP address and port number
ER	ercd	Error code

[Error Code]

E_OK	Normal completion
E_ID	Invalid ID number
E_NOEXS	Non-existent object
E_PAR	Parameter error (dstaddr address or tmout is invalid)
E_OBJ	Object state error (the specified TCP communication end point is already in use)
E_DLT	TCP reception point that is waiting for a connection request has been deleted
E_WBLK	Non-blocking call accepted
E_TMOUT	Polling failure or timeout
E_RLWAI	Forced release from waiting

[Functional Description]

This API makes the specified TCP reception point wait for a connection request. When a connection request comes, the connection processing is started at the specified TCP communication end point and the API returns the remote IP address and port number. The API remains in the waiting state until the connection processing is completed.

Multiple tcp_acp_cep calls can be issued to a single TCP reception point. How to choose a TCP communication end point for receiving the connection request in this case is implementation-dependent.

The TCP communication end point enters the passive-open-waiting state when a tcp_acp_cep call is issued, and enters the connected state when the API processing is completed. If the tcp_acp_cep processing is canceled due to a timeout or a tcp_can_cep call, the TCP communication end point will return to the unused state.

[Supplemental Information]

If a tcp_acp_cep call is issued to a TCP communication end point while another tcp_acp_cep processing is pending at the same end point, an E_OBJ error will be returned because the TCP communication end point is already in use.

tcp_con_cep Wait for Connection Request (Active Open)

[Standard Function]

[C Language API]

```
ER ercd = tcp_con_cep(ID cepid, T_IPV4EP *p_myaddr,
                      T_IPV4EP *p_dstaddr, TMO tmout);
```

[Parameter]

ID	cepid	TCP communication end point ID
T_IPV4EP	myaddr	Local IP address and port number
T_IPV4EP	dstaddr	Remote IP address and port number
TMO	tmout	Timeout

[Return Parameter]

ER	ercd	Error code

[Error Code]

E_OK	Normal completion
E_ID	Invalid ID number
E_NOEXS	Non-existent object
E_PAR	Parameter error (myaddr address, dstaddr address, IP address, port number, or tmout is invalid)
E_OBJ	Object state error (the specified TCP communication end point is already in use, the specified port number is already in use, or there is a privileged port violation)
E_WBLK	Non-blocking call accepted
E_TMOUT	Polling failure, timeout specified by tmout, or timeout determined by the protocol
E_RLWAI	Forced release from waiting
E_CLS	Connection request rejected

[Functional Description]

This API makes a connection between the specified TCP communication end point and the specified remote IP address and port number. This API remains in the waiting state until the connection processing is completed.

When IPV4_ADDRANY is specified as the local IP address or TCP_PORTANY is specified as the port number, the protocol stack should determine an IP address or a port number. When myaddr is set to NADR, the protocol stack should determine both IP address and port number. Details of privileged ports are implementation-dependent.

The TCP communication end point enters the active-open-waiting state when a tcp_con_cep call is issued, and enters the connected state when the API processing is completed. If the tcp_con_cep processing is canceled due to a timeout or a tcp_can_cep call, the TCP communication end point will return to the unused state.

[Supplemental Information]

If a tcp_con_cep call is issued to a TCP communication end point while another tcp_con_cep processing is pending at the same end point, an E_OBJ error will be returned because the TCP communication end point is already in use.

[Rationale]

Although polling specified as the timeout feature is not effective because the connection processing is never completed immediately, we decided not to actively eliminate the polling feature.

tcp_sht_cep Terminate Data Transmission [Standard Function]

[C Language API]

```
ER ercd = tcp_sht_cep(ID cepid);
```

[Parameter]

ID	cepid	TCP communication end point ID

[Return Parameter]

ER	ercd	Error code

[Error Code]

E_OK	Normal completion
E_ID	Invalid ID number
E_NOEXS	Non-existent object
E_OBJ	Object state error (the specified TCP communication end point is unconnected)

[Functional Description]

This API terminates data transmission at the specified TCP communication end point. Specifically, this API arranges the processing to send FIN and start the disconnection procedure after the data in the transmission buffer has been transmitted. As tcp_sht_cep only arranges the disconnection procedure, the processing never enters the waiting state within this API.

After a tcp_sht_cep call, the TCP communication end point is in the transmission-end state and no further data can be transmitted. If transmission is attempted in this state, an E_OBJ error will be returned. Data can still be received at the TCP communication end point.

[Supplemental Information]

If tcp_sht_cep is called multiple times for a single TCP communication end point, an E_OBJ error will be returned for the second and subsequent calls because the TCP communication end point is in the transmission-end state. If the pending state of the specified TCP communication end point needs to be indicated, a sub-error code should be used.

tcp_cls_cep Close Communication End Point [Standard Function]

[C Language API]

ER ercd = tcp_cls_cep(ID cepid, TMO tmout);

[Parameter]

ID	cepid	TCP communication end point ID
TMO	tmout	Timeout

[Return Parameter]

ER	ercd	Error code

[Error Code]

E_OK	Normal completion
E_ID	Invalid ID number
E_NOEXS	Non-existent object
E_PAR	Parameter error (tmout is invalid)
E_OBJ	Object state error (the specified TCP communication end point is unconnected)
E_WBLK	Non-blocking call accepted
E_TMOUT	Polling failure or timeout
E_RLWAI	Forced release from waiting

[Functional Description]

This API makes a disconnection for the TCP communication end point with the specified ID by sending FIN after waiting until the data in the transmit buffer has been transmitted. After the TCP communication end point enters the unused state, the data remaining in the receive buffer and the incoming data after that is discarded. As tcp_cls_cep returns only after the TCP communication end point enters the unused state, the TCP communication end point is ready for another purpose immediately after the return.

If the tcp_cls_cep processing is canceled due to a timeout or a tcp_can_cep call, it makes a forced disconnection by sending RST through the specified TCP communication end point and returns E_TMOUT or E_RLWAI, respectively. If RST cannot be sent immediately, this API only arranges RST transmission. If not even this arrangement is possible, RST transmission is omitted. In either case, the TCP communication end point is in the unused state when tcp_cls_cep returns (when the callback routine for reporting the completion is called if the non-blocking feature is specified).

[Supplemental Information]

Unlike the file descriptor in the socket interface, the communication end point of this specification does not become ready for another purpose until the TCP connection is completely terminated. Terminating the connection completely may take several minutes according to the TCP/IP protocol standard.

With a straightforward implementation of tcp_cls_cep, tcp_cls_cep does not return while the TCP port is in the TIMED-WAIT state in the protocol. If this implementation is adopted in a server for example, the server is unresponsive for a long time. An effective implementation to avoid this problem is to copy the TIMED-WAIT port (data structure for the communication end point) to another data structure for separate management, release the communication end point (data structure), and make tcp_cls_cep immediately return.

If a tcp_cls_cep call is issued to a TCP communication end point while another tcp_cls_cep processing is pending at the same end point, an E_OBJ error will be returned because the TCP communication end point is unconnected.

tcp_cls_cep is an exception to the principle that the object state remains unchanged upon returning from the timed-out API.

2.5 Transmitting and Receiving Data (Standard APIs)

tcp_snd_dat Send Data [Standard Function]

[C Language API]

ER ercd = tcp_snd_dat(ID cepid, VP data, INT len, TMO tmout);

[Parameter]

ID	cepid	TCP communication end point ID
VP	data	Start address of the data to be transmitted
INT	len	Length of the data to be transmitted
TMO	tmout	Timeout

[Return Parameter]

ER	ercd	Length of the data stored in the transmit buffer/error code

[Error Code]

Positive value	Normal completion (length of the data stored in the transmit buffer)
E_ID	Invalid ID number
E_NOEXS	Non-existent object
E_PAR	Parameter error (data, len, or tmout is invalid)
E_OBJ	Object state error (the specified TCP communication end point is in the unconnected or transmission-end state, or tcp_snd_dat or tcp_get_buf is pending)
E_WBLK	Non-blocking call accepted
E_TMOUT	Polling failure or timeout
E_RLWAI	Forced release from waiting
E_CLS	TCP connection disconnected

[Functional Description]

This API transmits data through the specified TCP communication end point. This API returns when data is stored in the transmit buffer. If the free area in the transmit buffer is shorter than the length of the data to be transmitted, this API stores the amount of data that fits into the available area and returns the length of the stored data. If the transmit buffer has no free area, this API remains in the waiting state until a free area becomes available.

tcp_snd_dat might send only part of the specified data and return a smaller byte count than the specified size. In contrast, the send function in the socket interface blocks other processed until all data has been transmitted.

If a tcp_snd_dat call is issued to a TCP communication end point while a tcp_snd_dat or

tcp_get_buf processing is pending at the same end point, an E_OBJ error will be returned.

tcp_rcv_dat Receive Data [Standard Function]

[C Language API]

ER ercd = tcp_rcv_dat(ID cepid, VP data, INT len, TMO tmout);

[Parameter]

ID	cepid	TCP communication end point ID
VP	data	Start address of the area to store the received data
INT	len	Length of the data to be received
TMO	tmout	Timeout

[Return Parameter]

ER	ercd	Length of the read data/error code

[Error Code]

Positive value	Normal completion (length of the read data)
0	End of data (correctly disconnected)
E_ID	Invalid ID number
E_NOEXS	Non-existent object
E_PAR	Parameter error (data, len, or tmout is invalid)
E_OBJ	Object state error (the specified TCP communication end point is unconnected, or tcp_rcv_dat or tcp_rcv_buf is pending)
E_WBLK	Non-blocking call accepted
E_TMOUT	Polling failure or timeout
E_RLWAI	Forced release from waiting
E_CLS	TCP connection disconnected and receive buffer is empty

[Functional Description]

This API receives data through the specified TCP communication end point. This API ends operation when data is read from the receive buffer. If the data stored in the receive buffer is shorter than the expected length of the data to be received, this API reads data until the receive buffer becomes empty and returns the length of the read data. If the receive buffer has no data, this API remains in the waiting state until data is received. After the connection is correctly disconnected by the remote host and all data is read from the receive buffer, this API returns 0.

If a tcp_rcv_dat call is issued to a TCP communication end point while a tcp_rcv_dat or tcp_rcv_buf processing is pending at the same end point, an E_OBJ error will be returned.

[Supplemental Information]

Even after the TCP connection is abnormally disconnected, data can be read by a tcp_rcv_dat call as long as the receive buffer has data.

2.6 Transmitting and Receiving Data (Copy-Saving APIs)

tcp_get_buf Get Transmit Buffer [Standard Function]

[C Language API]

ER ercd = tcp_get_buf(ID cepid, VP *p_buf, TMO tmout);

[Parameter]

ID	cepid	TCP communication end point ID
TMO	tmout	Timeout

[Return Parameter]

VP	buf	Start address of the free area
ER	ercd	Length of the free area/error code

[Error Code]

Positive value	Normal completion (length of the free area)
E_ID	Invalid ID number
E_NOEXS	Non-existent object
E_PAR	Parameter error (p_buf or tmout is invalid)
E_OBJ	Object state error (the specified TCP communication end point is in the unconnected or transmission-end state, or tcp_snd_dat or tcp_get_buf is pending)
E_WBLK	Non-blocking call accepted
E_TMOUT	Polling failure or timeout
E_RLWAI	Forced release from waiting
E_CLS	TCP connection disconnected

[Functional Description]

This API returns the start address of a free area in the transmit buffer where the next data for transmission can be stored and also returns the length of the continuous free area starting from the returned address. If the transmit buffer has no free area, this API remains in the waiting state until a free area becomes available.

As the internal state of the protocol stack remains unchanged after a tcp_get_buf call, the same area is returned if tcp_get_buf is called repeatedly in succession (the length of the free area might become longer). In contrast, the internal state of the protocol stack is changed by a tcp_snd_dat or tcp_snd_buf call. After either of these APIs is called, the information returned by a tcp_get_buf that was called previously is invalid.

If a tcp_get_buf call is issued to a TCP communication end point while a tcp_snd_dat or tcp_get_buf processing is pending at the same end point, an E_OBJ error will be

returned.

tcp_snd_buf Send Data from Buffer [Standard Function]

[C Language API]

ER ercd = tcp_snd_buf(ID cepid, INT len);

[Parameter]

ID	cepid	TCP communication end point ID
INT	len	Length of the data to be transmitted

[Return Parameter]

ER	ercd	Error code

[Error Code]

E_OK	Normal completion
E_ID	Invalid ID number
E_NOEXS	Non-existent object
E_PAR	Parameter error (len is invalid)
E_OBJ	Object state error (the specified TCP communication end point is in the unconnected or transmission-end state, or len is too long)
E_CLS	TCP connection disconnected

[Functional Description]

This API arranges the processing to transmit, for the length specified by len, the data stored in the buffer that was acquired by a tcp_get_buf call. As tcp_snd_buf only arranges transmission, the processing never enters the waiting state within this API.

[Supplemental Information]

The error code should be E_OBJ if the specified len exceeds the length of the continuous free area to store the transmit data (the length acquired by a tcp_snd_buf call), or E_PAR if the len value is not appropriate for any condition (a negative value, for example), in principle.

tcp_rcv_buf Receive Buffer Containing Data [Standard Function]

[C Language API]

ER ercd = tcp_rcv_buf(ID cepid, VP *p_buf, TMO tmout);

[Parameter]

ID	cepid	TCP communication end point ID

TMO	tmout	Timeout

[Return Parameter]

VP	buf	Start address of the received data
ER	ercd	Length of the received data/error code

[Error Code]

Positive value	Normal completion (length of the received data)
0	End of data (correctly disconnected)
E_ID	Invalid ID number
E_NOEXS	Non-existent object
E_PAR	Parameter error (p_buf or tmout is invalid)
E_OBJ	Object state error (the specified TCP communication end point is unconnected, or tcp_rcv_dat or tcp_rcv_buf is pending)
E_TMOUT	Polling failure or timeout
E_RLWAI	Forced release from waiting
E_CLS	TCP connection disconnected and receive buffer is empty

[Functional Description]

This API returns the start address of the buffer that contains received data and also returns the length of continuous data starting from the returned address. If the receive buffer has no data, this API remains in the waiting state until data is received. After the connection is correctly disconnected by the remote host and all data is read from the receive buffer, this API returns 0.

As the internal state of the protocol stack remains unchanged after a tcp_rcv_buf call, the same area is returned if tcp_rcv_buf is called repeatedly in succession (the length of the data might become longer). In contrast, the internal state of the protocol stack is changed by a tcp_rcv_dat or tcp_rel_buf call. After either of these APIs is called, the information returned by a tcp_rcv_buf that was called previously is invalid.

If a tcp_rcv_buf call is issued to a TCP communication end point while a tcp_rcv_dat or tcp_rcv_buf processing is pending at the same end point, an E_OBJ error will be returned.

[Supplemental Information]

Even after the TCP connection is abnormally disconnected, the start address and length of the received data can be acquired by a tcp_rcv_buf call as long as the receive buffer has data.

tcp_rel_buf Release Receive Buffer [Standard Function]

[C Language API]

```
ER ercd = tcp_rel_buf(ID cepid, INT len);
```

[Parameter]

ID	cepid	TCP communication end point ID
INT	len	Length of the data

[Return Parameter]

ER	ercd	Error code

[Error Code]

E_OK	Normal completion
E_ID	Invalid ID number
E_NOEXS	Non-existent object
E_PAR	Parameter error (len is invalid)
E_OBJ	Object state error (the specified TCP communication end point is unconnected or len is too long)

[Functional Description]

This API discards, for the length specified by len, the data in the buffer acquired by a tcp_rcv_buf call. The processing never enters the waiting state within this API.

[Supplemental Information]

The error code should be E_OBJ if the specified len exceeds the length of the received data stored in the continuous area (the length acquired by a tcp_rcv_buf call), or E_PAR if the len value is not appropriate for any condition (a negative value, for example), in principle.

2.7 Transmitting and Receiving Urgent Data

- In this API specification, urgent data should be handled as out-of-band data in a standard way, but this specification also allows an additional implementation-dependent way of handling it as in-band data by specifying attributes and options for TCP communication end points.

tcp_snd_oob Send Urgent Data [Extended Function]

[C Language API]

ER ercd = tcp_snd_oob(ID cepid, VP data, INT len, TMO tmout);

[Parameter]

ID	cepid	TCP communication end point ID
VP	data	Start address of the data to be transmitted
INT	len	Length of the data to be transmitted
TMO	tmout	Timeout

[Return Parameter]

ER	ercd	Length of the data stored in the transmit buffer/error code

[Error Code]

Positive value	Normal completion (length of the data stored in the transmit buffer)
E_ID	Invalid ID number
E_NOEXS	Non-existent object
E_PAR	Parameter error (data, len, or tmout is invalid)
E_OBJ	Object state error (the specified TCP communication end point is in the unconnected or transmission-end state, or tcp_snd_oob is pending)
E_WBLK	Non-blocking call accepted
E_TMOUT	Polling failure or timeout
E_RLWAI	Forced release from waiting
E_CLS	TCP connection disconnected

[Functional Description]

This API transmits urgent data through the specified TCP communication end point. This API remains in the waiting state until the data is stored in the transmit buffer.

If a tcp_snd_oob call is issued to a TCP communication end point while another tcp_snd_oob processing is pending at the same end point, an E_OBJ error will be returned.

tcp_rcv_oob Receive Urgent Data [Extended Function]

[C Language API]

```
ER ercd = tcp_rcv_oob(ID cepid, VP data, INT len);
```

[Parameter]

ID	cepid	TCP communication end point ID
VP	data	Start address of the area to store the received data
INT	len	Length of the area to store the received data

[Return Parameter]

ER	ercd	Length of the read data/error code

[Error Code]

Positive value	Normal completion (length of the read data)
E_ID	Invalid ID number
E_NOEXS	Non-existent object
E_PAR	Parameter error (data or len is invalid)
E_OBJ	Object state error (the specified TCP communication end

	point is unconnected, or urgent data has not been received)
E_BOVR	Buffer overflow

[Functional Description]

This API reads the urgent data that was received through the specified TCP communication end point and returns the length of the read data. If the length of the area to store the received data is shorter than the length of the received urgent data, this API stores the amount of data that fits into the available area, discards the remaining data, and returns E_BOVR. If no urgent data has been received, it will return an E_OBJ error.

[Supplemental Information]

This API is assumed to be called from the callback routine for receiving urgent data.

2.8 Other APIs

tcp_can_cep Cancel Pending Processing [Standard Function]

[C Language API]

```
ER ercd = tcp_can_cep(ID cepid, FN fncd);
```

[Parameter]

ID	cepid	TCP communication end point ID
FN	fncd	Function code of the API to be canceled

[Return Parameter]

ER	ercd	Error code

[Error Code]

E_OK	Normal completion
E_ID	Invalid ID number
E_NOEXS	Non-existent object
E_PAR	Parameter error (fncd is invalid)
E_OBJ	Object state error (the processing specified by fncd is not pending at the specified TCP communication end point)

[Functional Description]

This API cancels the specified processing pending at the specified TCP communication end point. An E_RLWAI error will be returned to the canceled API. When a non-blocking call is canceled, the callback routine for reporting the end of processing will be called.

The following is a list of the API names and function codes of the processing that can be canceled. When TFN_TCP_ALL is specified, all processing pending at the specified TCP communication end point can be canceled.

API Name	Function code
tcp_acp_cep	TFN_TCP_ACP_CEP
tcp_con_cep	TFN_TCP_CON_CEP
tcp_cls_cep	TFN_TCP_CLS_CEP
tcp_snd_dat	TFN_TCP_SND_DAT
tcp_rcv_dat	TFN_TCP_RCV_DAT
tcp_get_buf	TFN_TCP_GET_BUF
tcp_rcv_buf	TFN_TCP_RCV_BUF
tcp_snd_oob	TFN_TCP_SND_OOB
All	TFN_TCP_ALL

tcp_set_opt Set TCP Communication End Point Option

[Extended Function]

[C Language API]

ER ercd = tcp_set_opt(ID cepid, INT optname, VP optval, INT optlen);

[Parameter]

ID	cepid	TCP communication end point ID
INT	optname	Option type
VP	optval	Address of the area that stores the option value
INT	optlen	Length of the option value

[Return Parameter]

ER	ercd	Error code

[Error Code]

E_OK	Normal completion
E_ID	Invalid ID number
E_NOEXS	Non-existent object
E_PAR	Parameter error (optname, optval, or optlen is invalid)
E_OBJ	Object state error (specified option cannot be set in the current state)

[Functional Description]

This API sets the option value (equivalent to the socket option) for the specified TCP communication end point. optname specifies the option type to be set, optval specifies the address of the area that stores the option value, and optlen specifies the length of the option value. The types and functions of the options that can be specified for TCP communication end points are implementation-dependent.

[Supplemental Information]

This API should be used to provide a facility to specify IP options.

[Rationale]

For TCP reception points, no APIs are provided to set or read options because they are not considered necessary in most cases. When static option settings are required for TCP reception point creation, TCP reception point attributes can be used; for example, the TCP reception point attribute can be used to implement a facility equivalent to the SO_REUSEADDR socket option that allows reuse of a local IP address or port number.

tcp_get_opt Get TCP Communication End Point Option [Extended Function]

[C Language API]

ER ercd = tcp_get_opt(ID cepid, INT optname, VP optval, INT optlen);

[Parameter]

ID	cepid	TCP communication end point ID
INT	optname	Option type
VP	optval	Address of the area to store the option value
INT	optlen	Length of the area to store the option value

[Return Parameter]

ER	ercd	Length of the read option value/error code

[Error Code]

0 or a positive value	Normal completion (length of the read option value)
E_ID	Invalid ID number
E_NOEXS	Non-existent object
E_PAR	Parameter error (optname, optval, or optlen is invalid, or the area to store the option value is too short)
E_OBJ	Object state error (the specified option cannot be read in the current state)

[Functional Description]

This API reads the option value (equivalent to the socket option) for the specified TCP communication end point. optname specifies the option type to be read, optval specifies the address of the area to store the option value, and optlen specifies the length of the area to store the option value. If the area to store the option value is shorter than the length of the option value to be read, an E_PAR error will be returned. The types and functions of the options that can be read for TCP communication end points are implementation-dependent.

[Supplemental Information]

This API should be used to provide a facility to read IP options. It should also be used to provide a facility to read the state of TCP communication end points for debugging purposes.

2.9 Callbacks

Common Specifications

[C Language API]

ER ercd = callback(ID cepid, FN fncd, VP p_parblk);

[Common Parameter]

ID	cepid	TCP communication end point ID
FN	fncd	Event type

[Specific Parameter]

VP	p_parblk	Address of the parameter block specific to the event

[Return Value]

Depends on the event type

[Description]

The ID of the TCP communication end point where an event has occurred, the event type, and the parameters specific to the event are passed to this callback routine. The parameter block specific to the event can be referred to only in the callback routine. Usage of the return value from the callback routine depends on the event type.

The following describes the standard callbacks. Support of callbacks for other events is implementation-dependent.

Reporting End of Non-Blocking Call [Standard Function]

[Event Type]

Function code of the ended API processing

[Specific Parameter]

(The following parameter is passed through a parameter block)

ER	ercd	Return value from API

[Return Value]

Not used

[Function of Callback]

This routine is called when the non-blocking call processing is completed or canceled. The return value from the API is passed through a parameter block. The other parameters returned from the API are stored in the areas specified when the API was called.

[Supplemental Information]

When the tcp_rcv_buf processing with a non-blocking feature is completed, for example, the length of the continuous received data stored in the buffer is passed through ercd and the start address of the buffer is stored in the area indicated by the p_buf parameter that was specified when tcp_rcv_buf was called.

Receiving Urgent Data [Extended Function]

[Event Type]

TEV_TCP_RCV_OOB

[Specific Parameter]

(The following parameter is passed through a parameter block)

INT	len	Length of urgent data

[Return Value]

Not used

[Function of Callback]

This routine is called when urgent data is received. The urgent data should be read by using tcp_rcv_oob in this callback routine (if urgent data is not read before the callback routine returns, the data is discarded).

Chapter 3 APIs for UDP

3.1 Overview

- UDP communication end points (abbreviated to cep) are available for use with UDP.
- This specification does not provide a facility to specify in advance the remote IP address and port number for a UDP communication end point (equivalent to the connected UDP socket in the socket interface).
- The packet sent to the broadcast address for an interface is broadcast through the interface. The packet sent to address 255.255.255.255 is broadcast through all interfaces that support the broadcast function.

[Rationale]

A facility to specify in advance the remote address and port number for a UDP communication end point is realizable as a library on top of the APIs of this specification. Accordingly, this specification does not provide this facility in order to simplify the APIs.

3.2 Creating and Deleting UDP Communication End Points

UDP_CRE_CEP Create UDP Communication End Point
Static API...[Standard Function]
udp_cre_cep Dynamic API...[Extended Function]

[Static API]

```
UDP_CRE_CEP(ID cepid, { ATR cepatr, { UP myipaddr, UH myportno },
                                    FP callback });
```

[C Language API]

```
ER ercd = udp_cre_cep(ID cepid, T_UDP_CCEP *pk_ccep);
```

[Parameter]

ID	cepid	UDP communication end point ID to be created
T_UDP_CCEP	*pk_ccep	UDP communication end point creation information
pk_ccep includes		
ATR	cepatr	UDP communication end point attribute
T_IPV4EP	myaddr	Local IP address (myipaddr) and port number (myportno)
FP	callback	Callback routine address. Specify NULL when not using a callback routine.

(Other implementation-dependent parameters may be added)

[Return Parameter]

ER	ercd	Error code

[Error Code]

E_OK	Normal completion
E_ID	Invalid ID number
E_RSATR	Reserved attribute
E_PAR	Parameter error (pk_ccep address, IP address, port number, or callback is invalid)
E_OBJ	Object state error (a UDP communication end point is already created for the specified ID, the specified port number is already in use, or there is a privileged port violation)

[Functional Description]

These APIs create a UDP communication end point with the specified ID. When IPV4_ADDRANY is specified as the local IP address, the UDP packets that are sent to all local IP addresses or broadcast are received, and the protocol stack should determine the local IP address to be contained in transmit packets. When UDP_PORTANY is specified as the port number, the protocol stack should determine a port number. Usage of the UDP communication end point attribute and details of privileged ports are implementation-dependent.

udp_del_cep Delete UDP Communication End Point

Dynamic API...[Extended Function]

[C Language API]

ER ercd = udp_del_cep(ID cepid);

[Parameter]

ID	cepid	UDP communication end point ID

[Return Parameter]

ER	ercd	Error code

[Error Code]

E_OK	Normal completion
E_ID	Invalid ID number
E_NOEXS	Non-existent object

[Functional Description]

This API deletes the UDP communication end point with the specified ID. An E_DLT error will be returned to the API that is waiting for data transmission/reception at the deleted UDP communication end point.

3.3 Transmitting and Receiving Data

udp_snd_dat Send Packet [Standard Function]

[C Language API]

```
ER ercd = udp_snd_dat(ID cepid, T_IPV4EP *p_dstaddr, VP data,
                      INT len, TMO tmout);
```

[Parameter]

ID	cepid	UDP communication end point ID
T_IPV4EP	dstaddr	Remote IP address and port number
VP	data	Start address of the packet to be transmitted
INT	len	Length of the packet to be transmitted
TMO	tmout	Timeout

[Return Parameter]

ER	ercd	Zero or a positive value that indicates the length of the data stored in the transmit buffer/error code

[Error Code]

E_ID	Invalid ID number
E_NOEXS	Non-existent object
E_PAR	Parameter error (dstaddr address, IP address, port number, data, len, or tmout is invalid)
E_OBJ	Object state error (udp_snd_dat is pending)
E_QOVR	Queuing overflow
E_DLT	UDP communication end point that is waiting for transmission has been deleted
E_WBLK	Non-blocking call accepted
E_TMOUT	Polling failure or timeout
E_RLWAI	Forced release from waiting

[Functional Description]

This API transmits a packet from the specified UDP communication end point to the specified remote IP address and port number. This API remains in the waiting state until data is stored in the transmit buffer.

Multiple udp_snd_dat calls can be issued to a single UDP communication end point. A limit can be placed on the maximum number of udp_snd_dat calls to be queued if the implementation requires. When such a limit is set, an E_QOVR error is returned if udp_snd_dat calls have exceeded the limit.

[Rationale]

Queuing multiple udp_snd_dat calls requires a memory area for the queue. If there is no limit on the queued calls, the maximum memory area cannot be controlled. To avoid

this, this specification allows a limitation on the queued calls depending on the necessity for the implementation.

[Supplemental Information]

As data is sent in packet units in UDP, note that the return value (length of the data stored in the transmit buffer) does not necessarily indicate the length of the data actually received by the destination.

udp_rcv_dat Receive Packet [Standard Function]

[C Language API]

```
ER ercd = udp_rcv_dat(ID cepid, T_IPV4EP *p_dstaddr, VP data,
                      INT len, TMO tmout);
```

[Parameter]

ID	cepid	UDP communication end point ID
VP	data	Start address of the area to store the received packet
INT	len	Length of the area to store the received packet
TMO	tmout	Timeout

[Return Parameter]

T_IPV4EP	dstaddr	Remote IP address and port number
ER	ercd	Length of the read data/error code

[Error Code]

Positive value	Normal completion (length of the read data)
E_ID	Invalid ID number
E_NOEXS	Non-existent object
E_PAR	Parameter error (dstaddr address, data, len, or tmout is invalid)
E_OBJ	Object state error (udp_rcv_dat is pending)
E_QOVR	Queuing overflow
E_DLT	UDP communication end point that is waiting for reception has been deleted
E_WBLK	Non-blocking call accepted
E_TMOUT	Polling failure or timeout
E_RLWAI	Forced release from waiting
E_BOVR	Buffer overflow

[Functional Description]

This API receives a packet through the specified UDP communication end point and returns the remote IP address and port number. If no packet has been received, this API remains in the waiting state until a packet is received. If the area to store the received packet is shorter than the length of the received packet, this API stores the amount of

data that fits into the available area, discards the remaining data, and returns E_BOVR.

Multiple udp_rcv_dat calls can be issued to a single UDP communication end point. A limit can be placed on the maximum number of udp_rcv_dat calls to be queued if the implementation requires. When such a limit is set, an E_QOVR error is returned if udp_rcv_dat calls have exceeded the limit.

[Rationale]

Queuing multiple udp_rcv_dat calls requires a memory area for the queue. If there is no limit on the queued calls, the maximum memory area cannot be controlled. To avoid this, this specification allows a limitation on the queued calls depending on the necessity for the implementation.

3.4 Other APIs

udp_can_cep Cancel Pending Processing [Standard Function]

[C Language API]

ER ercd = udp_can_cep(ID cepid, FN fncd);

[Parameter]

ID	cepid	UDP communication end point ID
FN	fncd	Function code of the API to be canceled

[Return Parameter]

ER	ercd	Error code

[Error Code]

E_OK	Normal completion
E_ID	Invalid ID number
E_NOEXS	Non-existent object
E_PAR	Parameter error (fncd is invalid)
E_OBJ	Object state error (the processing specified by fncd is not pending at the specified UDP communication end point)

[Functional Description]

This API cancels the specified processing pending at the specified UDP communication end point. An E_RLWAI error will be returned to the canceled API. When a non-blocking call is canceled, the callback routine for reporting the end of processing will be called.

The following is a list of the API names and function codes of the processing that can be canceled. When TFN_UDP_ALL is specified, all processing pending at the specified UDP communication end point can be canceled.

API Name	Function code
udp_snd_dat	TFN_UDP_SND_DAT
udp_rcv_dat	TFN_UDP_RCV_DAT
All	TFN_UDP_ALL

udp_set_opt Set UDP Communication End Point Option [Extended Function]

[C Language API]

ER ercd = udp_set_opt(ID cepid, INT optname, VP optval, INT optlen);

[Parameter]

ID	cepid	UDP communication end point ID
INT	optname	Option type
VP	optval	Address of the area that stores the option value
INT	optlen	Length of the option value

[Return Parameter]

ER	ercd	Error code

[Error Code]

E_OK	Normal completion
E_ID	Invalid ID number
E_NOEXS	Non-existent object
E_PAR	Parameter error (optname, optval, or optlen is invalid)
E_OBJ	Object state error (specified option cannot be set in the current state)

[Functional Description]

This API sets the option value (equivalent to the socket option) for the specified UDP communication end point. optname specifies the option type to be set, optval specifies the address of the area that stores the option value, and optlen specifies the length of the option value. The types and functions of the options that can be specified for UDP communication end points are implementation-dependent.

[Supplemental Information]

This API should be used to provide a facility to specify IP options.

udp_get_opt Get UDP Communication End Point Option

[Extended Function]

[C Language API]

ER ercd = udp_get_opt(ID cepid, INT optname, VP optval, INT optlen);

[Parameter]

ID	cepid	UDP communication end point ID
INT	optname	Option type
VP	optval	Address of the area to store the option value
INT	optlen	Length of the area to store the option value

[Return Parameter]

ER	ercd	Length of the read option value/error code

[Error Code]

0 or a positive value	Normal completion (length of the read option value)
E_ID	Invalid ID number

E_NOEXS	Non-existent object
E_PAR	Parameter error (optname, optval, or optlen is invalid, or the area to store the option value is too short)
E_OBJ	Object state error (the specified option cannot be read in the current state)

[Functional Description]

This API reads the option value (equivalent to the socket option) for the specified UDP communication end point. optname specifies the option type to be read, optval specifies the address of the area to store the option value, and optlen specifies the length of the area to store the option value. If the area to store the option value is shorter than the length of the option value to be read, an E_PAR error will be returned. The types and functions of the options that can be read for UDP communication end points are implementation-dependent.

[Supplemental Information]

This API should be used to provide a facility to read IP options. It should also be used to provide a facility to read the state of UDP communication end points for debugging purposes.

3.5 Callbacks

Common Specifications

[C Language API]

ER ercd = callback(ID cepid, FN fncd, VP p_parblk);

[Common Parameter]

ID	cepid	UDP communication end point ID
FN	fncd	Event type

[Specific Parameter]

VP	p_parblk	Address of the parameter block specific to the event

[Return Value]

Depends on the event type

[Description]

The ID of the UDP communication end point where an event has occurred, the event type, and the parameters specific to the event are passed to a callback routine. The parameter block specific to the event can be referred to only in the callback routine. Usage of the return value from the callback routine depends on the event type.

The following describes the standard callbacks. Support of callbacks for other events is implementation-dependent.

Reporting End of Non-Blocking Call [Standard Function]

[Event Type]

Function code of the ended API processing

[Specific Parameter]

(The following parameter is passed through a parameter block)

ER	ercd	Return value from API

[Return Value]

Not used

[Function of Callback]

This routine is called when the non-blocking call processing is completed or canceled. The return value from the API is passed through a parameter block. The other parameters returned from the API are stored in the areas specified when the API was called.

[Supplemental Information]

When the udp_rcv_dat processing with a non-blocking feature is completed, for example, the length of the read data stored in the buffer is passed through ercd and the remote IP address and port number are stored in the area indicated by the p_dstaddr parameter that was specified when udp_rcv_dat was called.

Receiving UDP Packet [Standard Function]

[Event Type]

TEV_UDP_RCV_DAT

[Specific Parameter]

(The following parameter is passed through a parameter block)

INT	len	Length of packet

[Return Value]

Not used

[Function of Callback]

This routine is called when a UDP packet is received while udp_rcv_dat is not pending. The received UDP packet should be read by using udp_rcv_dat in the callback routine (if the packet is not read before the callback routine returns, the data is discarded).

Chapter 4 APIs for TCP/IPv6

4.1 Overview

- This chapter describes the API specification supporting IPv6. This specification was developed based on the API design policies and specifications supporting IPv4, which were described in Version 1.0.
- As this specification is an extension of the specification described in chapters 2 and 3, this chapter only describes the differences between the APIs for IPv4 and IPv6.

4.2 General Definitions

4.2.1 Data Structures and Data Types

(1) Data Structures for Containing an IP Address and a Port Number

In the following data structure for containing an IP address and a port number that was described in section 1.5.1,

```
typedef struct t_ipv4ep {
    UW          ipaddr;     /* IP address */
    UH          portno;     /* Port number */
} T_IPV4EP;
```

the representation of the IP address should be changed. To represent an IP address for IPv6, the T_IN6_ADDR data structure is defined as follows:

```
typedef struct t_in6_addr {
    union {
        UB __u6_addr8[16];
        UH __u6_addr16[8];
        UW __u6_addr32[4];
    };  __u6_addr;
} T_IN6_ADDR;
```

Using this definition, the T_IPV6EP data structure for IPv6 is defined as follows:

```
typedef struct t_ipv6ep {
    T_IN6_ADDR  ipaddr;   /* IP address */
    UH          portno;   /* Port number */
} T_IPV6EP;
```

[Supplemental Information]

An IP address and a port number should be assigned in the network order (big endian) to the ipaddr field and portno field, respectively.

(2) Data Structure for Creating Objects

The following T_TCP_CREP data structure for creating objects described in section 1.5.1 have a general name, but the data structure for containing an IP address and a port number for IPv4 is used as a member of T_TCP_CREP.

```
typedef struct t_tcp_crep {
    ATR         repatr;     /* TCP reception point attribute */
    T_IPV4EP    myaddr;     /* Local IP address and port number */
     (Other implementation-dependent fields may be added)
} T_TCP_CREP;
```

To adapt this T_TCP_CREP definition to IPv6, a new data structure for IPv6 is defined as follows:

```
typedef struct t_tcp_crep_ipv6 {
    ATR         repatr;     /* TCP reception point attribute */
    T_IPV6EP    myaddr;     /* Local IP address and port number */
} T_TCP_CREP_IPV6;
```

In the same way, the T_UDP_CCEP data structure for creating objects described in section 1.5.1 have a general name, but the data structure for containing an IP address and a port number for IPv4 is used as a member of T_UDP_CCEP.

```
typedef struct t_udp_ccep {
    ATR         cepatr;     /* UDP communication end point attribute */
    T_IPV4EP    myaddr;     /* Local IP address and port number */
    FP          callback;   /* Callback routine */
     (Other implementation-dependent fields may be added)
} T_UDP_CCEP;
```

T_UDP_CCEP_IPV6 is defined as a new data structure for IPv6 as follows:

```
typedef struct t_udp_ccep_ipv6 {
    ATR         cepatr;     /* UDP communication end point attribute */
    T_IPV6EP    myaddr;     /* Local IP address and port number */
    FP          callback;   /* Callback routine */
     (Other implementation-dependent fields may be added)
} T_UDP_CCEP_IPV6;
```

(3) Special IP Address and Port Numbers

As described in item (5), Special IP Address and Port Numbers, in section 1.5.3, Constants, IPV4_ADDRANY is a constant for IPv4.

IPV4_ADDRANY0 IP address specification omitted

The equivalent data for IPv6 is defined as follows:

```
#define IPV6_ADDR_UNSPECIFIED_INIT \
    {{ 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, \
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 }}
#define IPV6_ADDRANY    IPV6_ADDR_UNSPECIFIED_INIT
```

In addition, the protocol stack defines data CONST_IPV6_ADDR_ANY as follows so that the user can choose a convenient one in the programs.

```
const struct i_in6_addr    __ipv6_addr_unspecified_init =
    {{ 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, \
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 }}
#define CONST_IPV6_ADDRANY    __ipv6_addr_unspecified
```

(4) Creating Reception Points

As described in section 2.2, Creating and Deleting TCP Reception Points, the TCP_CRE_REP static API has a general name, but it includes an address specification for IPv4.

[Static API]

```
TCP_CRE_REP(ID repid, { ATR repatr, { UP myipaddr, UH myportno }});
```

To adapt it to IPv6, the following API is defined.

```
TCP_CRE_REP_IPV6(ID repid, { ATR repatr, { T_IN6_ADDR myipaddr,
                                    UH myportno }});
```

(5) Connecting and Disconnecting

As described in section 2.4, Connecting and Disconnecting, the TCP_ACP_CEP API for waiting for connection requests and the TCP_CON_CEP API for requesting connection have general names, but they include an address specification for IPv4.

[C Language API]

```
ER ercd = tcp_acp_cep(ID cepid, ID repid, T_IPV4EP *p_dstaddr, TMO
                                    tmout);
ER ercd = tcp_con_cep(ID cepid, T_IPV4EP *p_myaddr,
                                    T_IPV4EP *p_dstaddr, TMO tmout);
```

To adapt them to IPv6, the following APIs are defined.

```
ER ercd = tcp_acp_cep_ipv6(ID cepid, ID repid, T_IPV6EP *p_dstaddr,
                                        TMO tmout);
ER ercd = tcp_con_cep_ipv6(ID cepid, T_IPV6EP *p_myaddr,
                                        T_IPV6EP *p_dstaddr, TMO tmout);
```

(6) Creating and Deleting UDP Communication End Points

As described in section 3.2, Creating and Deleting UDP Communication End Points, the UDP_CRE_REP static API for creating a UDP communication end point has a general name, but it includes an address specification for IPv4 in the same way as TCP_CRE_REP does for creating a TCP reception point.

[Static API]

```
UDP_CRE_CEP(ID cepid, { ATR cepatr, { UP myipaddr, UH myportno },
                                        FP callback });
```

To adapt it to IPv6, the following API is defined.

```
UDP_CRE_CEP_IPV6(ID cepid, { ATR cepatr, { T_IN6_ADDR myipaddr,
                                        UH myportno },
                                        FP callback });
```

(7) Transmitting and Receiving Data

As described in section 3.3, Transmitting and Receiving Data, the UDP_SND_DAT API for transmitting a packet and the UDP_RCV_DAT API for receiving a packet have general names, but they include address specifications for IPv4 in the same way as TCP_ACP_CEP does for waiting for connection requests.

[C Language API]

```
ER ercd = udp_snd_dat(ID cepid, T_IPV4EP *p_dstaddr, VP data,
                                        INT len, TMO tmout);
ER ercd = udp_rcv_dat(ID cepid, T_IPV4EP *p_dstaddr, VP data,
                                        INT len, TMO tmout);
```

To adapt them to IPv6, the following APIs are defined.

```
ER ercd = udp_snd_dat_ipv6(ID cepid, T_IPV6EP *p_dstaddr, VP data,
                                        INT len, TMO tmout);
ER ercd = udp_rcv_dat_ipv6(ID cepid, T_IPV6EP *p_dstaddr, VP data,
                                        INT len, TMO tmout);
```

Chapter 5 Conditions for Using the Specification

5.1 Open Specification, No Warranty, Copyright

The ITRON TCP/IP API Specification is an open specification handled according to the basic policy of the TRON Project. Anyone is free to implement, use, and sell, without paying a license fee, the TCP/IP protocol stack conforming to the API defined in this specification.

As an incorporated body, TRON Association makes no guarantees whatsoever with regard to this specification, including its correctness and validity. TRON Association is not liable for any direct or indirect loss or damage caused by using this specification.

5.2 Contact Information

The ITRON specifications are developed and maintained by the ITRON Specification Study Group of TRON Association. Any questions regarding the specifications should be directed to the following.

ITRON Specification Study Group, 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
E-mail: info@assoc.tron.org
Website: http://www.assoc.tron.org

Appendix A Version History

May 14, 1998	Ver. 1.00.00	First edition published
May 19, 1998	Ver. 1.00.01	Minor errors corrected (such as typographical errors)
April 1, 2006	Ver. 2.00.00	Specification adapted to the current TRON specification and new specification for IPv6 added

Editorial Committee Members for the English Edition of the ITRON TCP/IP API Specification Ver. 2.00.00:

Leader: Noriaki Takakura (NEC Electronics Corporation)
Tomonori Kaneko (eSOL Co., Ltd.)
Tadashi Sakamoto (Renesas Solutions Corp.)
Hiroshi Ii (TRON Association)
Shimpei Matsumura (TRON Association)

Appendix B Relations with the Socket Interface

B.1 Major Differences from the Socket Interface

The API of this specification has the following major differences from the socket interface.

- The socket interface specifies APIs independently of the protocols but this specification specifies separate APIs for each protocol; specifically, separate APIs for TCP and UDP. In addition, the address specification structure defined in this specification is specific to TCP and UDP on IPv4. This specification does not support other protocols.
- This specification uses communication end points as the abstraction equivalent to sockets. A separate type of communication end point is defined for each protocol and type. Specifically, this specification distinguishes communication end points for TCP from those for UDP, and further distinguishes the communication end points for TCP between TCP reception points and TCP communication end points. This specification provides a dedicated API for creating and deleting each type of communication end point.
- This specification does not provide the abstraction equivalent to file descriptors in the socket interface, but allows direct manipulation of communication end points through APIs (refer to section 1.4.3). This specification does not support a feature equivalent to the select function in the socket interface; a similar feature can be implemented through a callback.
- This specification provides a timeout feature and a non-blocking call feature for each API that may enter the waiting state. The socket interface also provides a non-blocking feature, but it differs from this specification in that the non-blocking feature should be specified for each socket instead of each API.
- This specification provides features for canceling the pending processing (tcp_can_cep and udp_can_cep).
- The listen function in the socket interface is included in the API for creating a TCP reception point (tcp_cre_rep), which should be used as the equivalent to listen. Note that tcp_cre_rep does not have a feature for specifying the maximum number of connection requests to be queued, which is supported by the listen function in the socket interface.
- The bind function in the socket interface is included in each of the APIs (tcp_cre_rep, tcp_con_cep, and udp_cre_cep); there is no dedicated API equivalent to bind.
- The accept function in the socket interface internally creates a socket dynamically when receiving a connection request. The tcp_acp_cep API in this specification passes the TCP communication end point to be used for connection through a parameter when receiving a connection request.

- This specification provides copy-saving APIs for TCP.
- This specification does not provide an API for terminating only reception through TCP connection. In the socket interface, whether to terminate transmission or reception can be specified as a parameter of the shutdown function, but tcp_sht_cep of this specification does not have an equivalent parameter.
- The APIs for transmitting and receiving TCP urgent data are separate from those for regular data transmission and reception in this specification. The received urgent data is discarded if it is not read within the callback routine; it is not queued in the protocol stack.
- This specification provides a callback routine to be called when a UDP packet is received. The received data is discarded if it is not read within the callback routine; it is not queued in the protocol stack.
- This specification does not support a feature equivalent to the connected UDP socket.

Appendix C Sample Programs

C.1 Sample Implementation of read and write by TCP Copy-Saving APIs

```
ER read(ID cepid, VP buf, INT len, TMO tmout)
{
        VP rbuf;
        INT rlen;
        INT tlen = 0;       /* Total length of received data */

        if ((rlen = tcp_rcv_buf(cepid, &rbuf, tmout)) <= 0) {
                return(rlen);
        }
        while (len > 0 && rlen > 0) {
            if (rlen > len) {
                        rlen = len;
            }
            bcopy(rbuf, buf, rlen);
            buf += rlen;
            len -= rlen;
            tlen += rlen;
            if (tcp_rel_buf(cepid, rlen) < 0) {
                        /* Returns the length of data received until an error occurs */
                        return(tlen);
            }
            if ((rlen = tcp_rcv_buf(cepid, &rbuf, TMO_POL)) < 0) {
                        /* Returns the length of data received until an error occurs */
                        return(tlen);
            }
        }
        return(tlen);
}

ER write(ID cepid, VP buf, INT len, TMO tmout)
{
        VP sbuf;
        INT slen;
        INT tlen = 0;       /* Total length of transmitted data */

        if ((slen = tcp_get_buf(cepid, &sbuf, tmout)) <= 0) {
                return(slen);
        }
        while (len > 0 && slen > 0) {
            if (slen > len) {
                        slen = len;
            }
            bcopy(buf, sbuf, slen);
            buf += slen;
            len -= slen;
```

```
                tlen += slen;
                if (tcp_snd_buf(cepid, slen) < 0) {
                        /* Returns the length of data transmitted until an error occurs */
                                return(tlen);
                }
                if ((slen = tcp_get_buf(cepid, &sbuf, TMO_POL)) < 0) {
                        /* Returns the length of data transmitted until an error occurs */
                                return(tlen);
                }
        }
        return(tlen);
}
```

C.2 Sample Implementation of getc and putc by TCP Copy-Saving APIs

- The target of transmission and reception is fixed at a single TCP communication end point.
- The processing of the errors returned by the APIs is omitted.

```
extern ID cepid;

static unsigned char *rcvbuf;
static INT rcvbuflen = 0;
static INT rcvdatlen = 0;

int getc()
{
        if (rcvbuflen == 0) {
                if (rcvdatlen > 0) {
                        tcp_rel_buf(cepid, rcvdatlen);
                        rcvdatlen = 0;
                }
                rcvbuflen = tcp_rcv_buf(cepid, &rcvbuf, TMO_FEVR);
                if (rcvbuflen == 0) {
                        /* End of data (EOF) */
                        return(-1);
                }
        }
        rcvbuflen -= 1;
        rcvdatlen += 1;
        return(*rcvbuf++);
}

static unsigned char *sndbuf;
static INT sndbuflen = 0;
static INT snddatlen = 0;

void putc(char c)
```

```
{
        if (sndbuflen == 0) {
            if (snddatlen > 0) {
                    tcp_snd_buf(cepid, snddatlen);
                    snddatlen = 0;
            }
            sndbuflen = tcp_get_buf(cepid, &sndbuf, TMO_FEVR);
        }
        sndbuflen -= 1;
        snddatlen += 1;
        *sndbuf++ = c;
}

void flush()
{
        if (snddatlen > 0) {
                tcp_snd_buf(cepid, snddatlen);
                snddatlen = 0;
                sndbuflen = 0;
        }
        if (rcvdatlen > 0) {
                tcp_rel_buf(cepid, rcvdatlen);
                rcvdatlen = 0;
                rcvbuflen = 0;
        }
}
```

C.3 Example of UDP Callback Routine

```
#define MY_IPADDR     (0xc0a80001)       /* 192.168.0.1 */
#define MY_PORTNO     htons(65500)
#define BUF_CNT       (8)
#define search_empty_buffer(index)      search_buffer(1, (index))
#define search_received_buffer(index)  search_buffer(0, (index))
typedef struct buffer {
        volatile INT    datsiz;     /* Data size in buffer */
        T_IPV4EP        dstaddr;    /* Destination address received from or sent to */
        char            buf[256];
} buffer_t;
static buffer_t _bufs[BUF_CNT];
int get_next_index(int current)
{
        return (current + 1 < BUF_CNT) ? current + 1 : 0;
}
int search_buffer(int empty, int current)
{
        int tmpidx;

        tmpidx = current;
        while (empty ? (0 != _bufs[tmpidx].datsiz) : (0 == _bufs[tmpidx].datsiz)) {
```

```
                tmpidx = get_next_index(tmpidx);
                if (tmpidx == current) {
                    /* Not found */
                    tmpidx = -1;
                    break;
                }
            }
            return tmpidx;
}
ER udp_callback(ID cepid, FN fncd, VP p_parblk)
{
            static int  cbidx = 0;
            int         tmpidx;
            int         rcvsiz;
            buffer_t    *p_buf;

            if (TEV_UDP_RCV_DAT == fncd) {
                tmpidx = search_empty_buffer(cbidx);
                if (tmpidx >= 0) {
                    cbidx = tmpidx;
                    p_buf = &_bufs[cbidx];
                    rcvsiz = *((INT *)p_parblk);
                    udp_rcv_dat(cepid, &p_buf->dstaddr, p_buf->buf, rcvsiz,
                                TMO_POL);
                    p_buf->datsiz = rcvsiz;
                    cbidx = get_next_index(cbidx);
                }
            }
            return 0;
}
void test_main()
{
            ER              ercd;
            ID              cepid;
            T_UDP_CCEP      ccep;
            buffer_t        *p_buf;
            int             sndidx;
            int             tmpidx;

            memset(&_bufs, 0, sizeof(_bufs));
            sndidx = 0;
            memset(&ccep, 0, sizeof(ccep));
            ccep.myaddr.ipaddr = MY_IPADDR;
            ccep.myaddr.portno = MY_PORTNO;
            ccep.callback = udp_callback;
            cepid = 1;
            ercd = udp_cre_cep(cepid, &ccep);
            while (ercd >= 0) {
                /* Loop back received data */
                tmpidx = search_received_buffer(sndidx);
```

```
                if (tmpidx >= 0) {
                        sndidx = tmpidx;
                        p_buf = &_bufs[sndidx];
                        ercd = udp_snd_dat(cepid, &p_buf->dstaddr, p_buf->buf,
                                                p_buf->datsiz, TMO_FEVR);
                if (ercd < 0) {
                        /* Exit from loop because an error occurred */
                        break;
                }
                p_buf->datsiz = 0;
                sndidx = get_next_index(sndidx);
            }
        }
        udp_del_cep(cepid);
}
```

Appendix D Type Definition Macros

D.1 Differences in Type Definition Macros

The type definition macros differ as follows according to the target OS.

Table 1.

Type definition	Function	µITRON V3	µITRON V4	T-Kernel V1.00
INT	Signed integer for the processor	Implementation-dependent	Implementation-dependent	typedef int INT
UH	Unsigned 16-bit integer	typedef unsigned short UH	Implementation-dependent	typedef unsigned short UH
UW	Unsigned 32-bit integer	typedef unsigned long UW	Implementation-dependent	typedef unsigned int UW
VP	Pointer to an unknown data type	typedef void *VP	Implementation-dependent	typedef void *VP
FP	Processing unit start address (pointer to a function)	typedef void (*FP)()	Implementation-dependent	typedef void (*FP)()
ID	Object ID number	Implementation-dependent signed integer	Implementation-dependent16-bit or longer signed integer	typedef INT ID
ER	Error code	Signed integer	Implementation-dependent 8-bit or longer signed integer	typedef INT ER
ATR	Object attribute	Unsigned integer	Unsigned integer	typedef UINT ATR
NADR	Invalid address	(-1)	Not defined	NULL should be used instead
SIZE	Memory area size	Not defined	Unsigned integer	Not defined
TMO	Timeout	Same as INT	Implementation-dependent 16-bit or longer signed integer	typedef INT TMO
FN	Function code	Signed integer of a maximum of two bytes	Signed integer	typedef INT FN

Appendix E Differences in Constants According to Target OS

E.1 Differences in Error Codes

The error codes differ as follows according to the target OS.

Table 2.

Main error code	µITRON V3	µITRON V4	T-Kernel V1.00
E_OK	0	0	0
E_SYS	(-5)	-5	ERCD(-5,0)
E_NOSPT	(-17)	-9	ERCD(-33,0)
E_RSATR	(-24)	-11	ERCD(-11,0)
E_PAR	(-33)	-17	ERCD(-17,0)
E_ID	(-35)	-18	ERCD(-18,0)
E_MACV	(-65)	-26	ERCD(-26,0)
E_NOMEM	(-10)	-33	ERCD(-33,0)
E_OBJ	(-63)	-41	ERCD(-41,0)
E_NOEXS	(-52)	-42	ERCD(-42,0)
E_QOVR	(-73)	-43	ERCD(-43,0)
E_RLWAI	(-86)	-49	ERCD(-49,0)
E_TMOUT	(-85)	-50	ERCD(-50,0)
E_DLT	(-81)	-51	ERCD(-51,0)
E_WBLK	(-83)*[1]	-57	ERCD(-57,0)*[1]
E_CLS	(-87)*[1]	-52	ERCD(-52,0)*[1]
E_BOVR	(-89)*[1]	-58	ERCD(-58,0)*[1]

- *1) These codes are not defined in the kernel specifications, but are defined in the ITRON TCP/IP API specification. Refer to section 1.5.3.

E.2 Differences in Timeout Specifications

The timeout specifications differ as follows according to the target OS.

Table 3.

Timeout specification	µITRON V3	µITRON V4	T-Kernel V1.00
TMO_POL	0	0	0
TMO_FEVR	-1	-1	-1
TMO_NBLK	-2*[1]	-2	-2*[1]

- *1) These values are not defined in the kernel specifications, but are defined in the ITRON TCP/IP API specification. Refer to section 1.5.3.

Appendix F Notes on Implementation

F.1 Implementation of ER Type

The ER type to be used for the return value from the API of this specification should be data of the same size as the INT type. Although there have been some implementations of the ER type in eight bits, the same size as the INT type (16 bits or 32 bits) is recommended.

F.2 NADR Value

Note the NADR value because it differs according to the target OS. The following shows the NADR value for each OS. Although the NADR value is not defined in the kernel specifications other than µITRON V3, it is defined in the ITRON TCP/IP API specification.

- µITRON V3: NADR = -1
- µITRON V4: NADR = NULL (= 0)
- T-Kernel V1.00: NADR = NULL (= 0)

Appendix G Differences between Specifications for IPv4 and IPv6

G.1 Differences in Specifications

The following table is a summary of the differences between the specifications for IPv4 and IPv6.

Table 4.

Meaning of structure definitions and functions	Specification for IPV4	Specification for IPV6
Name of the data structure containing an IP address and a port number	T_IPV4EP	T_IPV6EP
Name of the data structure for object creation	T_TCP_CREP	T_TCP_CREP_IPV6
Name of the data structure for object creation	T_UDP_CCEP	T_UDP_CCEP_IPV6
Definition of the value used when the IP specification is omitted	IPV4_ADDRANY	IPV6_ADDRANY CONST_IPV6_ADDRANY
API for creating a TCP reception point	TCP_CRE_REP tcp_cre_rep	TCP_CRE_REP_IPV6 tcp_cre_rep_ipv6
API for waiting for a connection request (passive open)	tcp_acp_cep	tcp_acp_cep_ipv6
API for requesting a connection (active open)	tcp_con_cep	tcp_con_cep_ipv6
API for creating a UDP communication end point	UDP_CRE_CEP udp_cre_cep	UDP_CRE_CEP_IPV6 udp_cre_cep_ipv6
API for transmitting a packet	udp_snd_dat	udp_snd_dat_ipv6
API for receiving a packet	udp_rcv_dat	udp_rcv_dat_ipv6

ITRON TCP/IP API Specification Version 2.00.00

July 24, 2007 Version 2.00.00 (English) First edition

Supervised by: Ken Sakamura

Edited and published by: TRON Association

Katsuta Building 5F

3-39, Mita 1-chome, Minato-ku, Tokyo 108-0073, JAPAN

TEL: +81-3-3454-3191

Printed by: Hokuetsu Printing CO., LTD.