BEA Tuxedo®

# ATMI C Function Reference

Version 9.1
Document Released: May 16, 2006

# Copyright



# Restricted Rights Legend



# Trademarks and Service Marks

# Contents

## About This Document



## Section 3c - C Functions

# About This Document

This document provides reference information on C language functions used in the BEA Tuxedo ATMI environment. The reference pages are arranged in alphabetical order by function name.

## What You Need to Know

This document is intended for the following audiences:

- administrators who are interested in configuring and managing applications in a BEA Tuxedo environment
- application developers who are interested in programming applications in a BEA Tuxedo environment

This document assumes a familiarity with the BEA Tuxedo platform and C programming.

## e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation or go directly to the "e-docs" Product Documentation page at http://e-docs.bea.com.

## How to Print the Document

You can print a copy of this document from a Web browser, one file at a time, by using the File—>Print option on your Web browser.

A PDF version of this document is available on the BEA Tuxedo documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the BEA Tuxedo documentation Home page, click the PDF files button and select the document you want to print.

If you do not have the Adobe Acrobat Reader, you can get it for free from the Adobe Web site at http://www.adobe.com/.

## Related Information

Related documents are listed in the See Also section of each reference page.

## Contact Us!

Your feedback on the BEA Tuxedo documentation is important to us. Send us e-mail at **docsupport@bea.com** if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the BEA Tuxedo documentation.

In your e-mail message, please indicate that you are using the documentation for the BEA Tuxedo 9.1 release.

If you have any questions about this version of BEA Tuxedo, or if you have problems installing and running BEA Tuxedo, contact BEA Customer Support through BEA WebSupport at **www.bea.com**. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

# Documentation Conventions

The following documentation conventions are used throughout this document.

| Convention | Item |
|---|---|
| **boldface text** | Indicates terms defined in the glossary. |
| Ctrl+Tab | Indicates that you must press two or more keys simultaneously. |
| *italics* | Indicates emphasis or book titles. |
| `monospace text` | Indicates code samples, commands and their options, data structures and their members, data types, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.<br>*Examples*:<br>`#include <iostream.h> void main ( ) the pointer psz`<br>`chmod u+w *`<br>`\tux\data\ap`<br>`.doc`<br>`tux.doc`<br>`BITMAP`<br>`float` |
| **`monospace boldface text`** | Identifies significant words in code.<br>*Example*:<br>`void` **`commit`** `( )` |
| *`monospace italic text`* | Identifies variables in code.<br>*Example*:<br>`String` *`expr`* |
| UPPERCASE TEXT | Indicates device names, environment variables, and logical operators.<br>*Example*s:<br>LPT1<br>SIGNON<br>OR |

| Convention | Item |
|---|---|
| `{ }` | Indicates a set of choices in a syntax line. The braces themselves should never be typed. |
| `[ ]` | Indicates optional items in a syntax line. The brackets themselves should never be typed.<br>*Example*:<br>`buildobjclient [-v] [-o name ] [-f` *`file-list`*`]...`<br>`[-l` *`file-list`*`]...` |
| \| | Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed. |
| `...` | Indicates one of the following in a command line:<br>• That an argument can be repeated several times in a command line<br>• That the statement omits additional optional arguments<br>• That you can enter additional parameters, values, or other information<br>The ellipsis itself should never be typed.<br>*Example*:<br>`buildobjclient [-v] [-o name ] [-f` *`file-list`*`]...`<br>`[-l` *`file-list`*`]...` |
| .<br>.<br>. | Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed. |

# Section 3c - C Functions

**Table 1 BEA Tuxedo ATMI C Functions**

| Name | Description |
|---|---|
| Introduction to the C Language Application-to-Transaction Monitor Interface | Provides an introduction to the C language ATMI |
| `AEMsetblockinghook(3c)` | Establishes an application-specific blocking hook function |
| `AEOaddtypesw(3c)` | Installs or replaces a user-defined buffer type at execution time |
| `AEPisblocked(3c)` | Determines if a blocking call is in progress |
| `AEWsetunsol(3c)` | Posts Windows message for BEA Tuxedo ATMI unsolicited event |
| `buffer(3c)` | Semantics of elements in `tmtype_sw_t` |
| `catgets(3c)` | Reads a program message |
| `catopen, catclose(3c)` | Opens/closes a message catalogue |
| `decimal(3c)` | Decimal conversion and arithmetic routines |
| `getURLEntityCacheDir(3c)` | Gets the absolute path to the location where the DTD, Schemas, and Entity files are cached. It specifies a particular Xerces parser class method. |

**Table 1 BEA Tuxedo ATMI C Functions (Continued)**

| Name | Description |
|---|---|
| getURLEntityCaching(3c) | Gets the caching mechanism for the DTD, schemas, and Entity files. It specifies a particular Xerces parser class method. |
| gp_mktime(3c) | Converts a tm structure to a calendar time |
| nl_langinfo(3c) | Language information |
| setlocale(3c) | Modifies and queries a program's locale |
| setURLEntityCacheDir(3c) | Sets the directory where the DTD, schemas, and Entity files are to be cached. It specifies a particular Xerces parser class method. |
| setURLEntityCaching(3c) | Turns caching on or off for DTD, schema, and Entity files by default. It specifies a particular Xerces parser class method. |
| strerror(3c) | Gets error message string |
| strftime(3c) | Converts date and time to string |
| tpabort(3c) | Routine for aborting current transaction |
| tpacall(3c) | Routine for sending a service request |
| tpadmcall(3c) | Administers unbooted application |
| tpadvertise(3c) | Routine for advertising a service name |
| tpalloc(3c) | Routine for allocating typed buffers |
| tpbegin(3c) | Routine for beginning a transaction |
| tpbroadcast(3c) | Routine to broadcast notification by name |
| tpcall(3c) | Routine for sending service request and awaiting its reply |
| tpcancel(3c) | Routine for canceling a call descriptor for outstanding reply |
| tpchkauth(3c) | Routine for checking if authentication required to join an application |
| tpchkunsol(3c) | Routine for checking for unsolicited message |

**Table 1 BEA Tuxedo ATMI C Functions (Continued)**

| Name | Description |
|---|---|
| tpclose(3c) | Routine for closing a resource manager |
| tpcommit(3c) | Routine for committing current transaction |
| tpconnect(3c) | Routine for establishing a conversational service connection |
| tpconvert(3c) | Converts structures to/from string representations |
| tpconvmb(3c) | Converts encoding of characters in an input buffer to a named target encoding |
| tpcryptpw(3c) | Encrypts application password in administrative request |
| tpdequeue(3c) | Routine to dequeue a message from a queue |
| tpdiscon(3c) | Routine for taking down a conversational service connection |
| tpenqueue(3c) | Routine to enqueue a message |
| tpenvelope(3c) | Accesses the digital signature and encryption information associated with a typed message buffer |
| tperrordetail(3c) | Gets additional detail about an error generated from the last BEA Tuxedo ATMI system call |
| tpexport(3c) | Converts a typed message buffer into an exportable, machine-independent string representation, that includes digital signatures and encryption seals |
| tpfml32toxml(3c) | Converts FML32 buffer data to XML buffer data |
| tpfmltoxml(3c) | Converts FML buffer data to XML buffer data |
| tpforward(3c) | Routine for forwarding a service request to another service routine |
| tpfree(3c) | Routine for freeing a typed buffer |
| tpgblktime(3c) | Routine for returning a previously set, per second nontransactional blocktime value |
| tpgetadmkey(3c) | Gets administrative authentication key |

**Table 1 BEA Tuxedo ATMI C Functions (Continued)**

| Name | Description |
|---|---|
| tpgetctxt(3c) | Retrieves a context identifier for the current application association |
| tpgetlev(3c) | Routine for checking if a transaction is in progress |
| tpgetmbenc(3c) | Gets the code-set encoding name from a typed buffer |
| tpgetrepos(3c) | Routine for retrieving service and parameter information from a Tuxedo repository file. |
| tpgetrply(3c) | Routine for getting a reply from a previous request |
| tpgprio(3c) | Routine for getting a service request priority |
| tpimport(3c) | Converts an exported representation back into a typed message buffer |
| tpinit(3c) | Joins an application |
| tpkey_close(3c) | Closes a previously opened key handle |
| tpkey_getinfo(3c) | Gets information associated with a key handle |
| tpkey_open(3c) | Opens a key handle for digital signature generation, message encryption, or message decryption |
| tpkey_setinfo(3c) | Sets optional attribute parameters associated with a key handle |
| tpnotify(3c) | Routine for sending notification by client identifier |
| tpopen(3c) | Routine for opening a resource manager |
| tppost(3c) | Posts an event |
| tprealloc(3c) | Routine to change the size of a typed buffer |
| tprecv(3c) | Routine for receiving a message in a conversational connection |
| tpresume(3c) | Resumes a global transaction |
| tpreturn(3c) | Routine for returning from a service routine |

**Table 1 BEA Tuxedo ATMI C Functions (Continued)**

| Name | Description |
|---|---|
| tpsblktime(3c) | Routine for setting nontransactional blocktime values, in seconds, for the next service call or for all service calls used per context |
| tpscmt(3c) | Routine for setting when tpcommit() should return |
| tpseal(3c) | Marks a typed message buffer for encryption |
| tpsend(3c) | Routine for sending a message in a conversational connection |
| tpservice(3c) | Template for service routines |
| tpsetctxt(3c) | Sets a context identifier for the current application association |
| tpsetmbenc(3c) | Sets the code-set encoding name for a typed buffer |
| tpsetrepos(3c) | Adds, edits, or deletes service and parameter information from a Tuxedo Service Metadata repository file |
| tpsetunsol(3c) | Sets the method for handling unsolicited messages |
| tpsign(3c) | Marks a typed message buffer for digital signature |
| tpsprio(3c) | Routine for setting service request priority |
| tpstrerror(3c) | Gets error message string for a BEA Tuxedo ATMI system error |
| tpstrerrordetail(3c) | Gets error detail message string for a BEA Tuxedo ATMI system |
| tpsubscribe(3c) | Subscribes to an event |
| tpsuspend(3c) | Suspends a global transaction |
| tpsvrdone(3c) | Terminates a BEA Tuxedo ATMI system server |
| tpsvrinit(3c) | Initializes a BEA Tuxedo ATMI system server |
| tpsvrthrdone(3c) | Terminates a BEA Tuxedo ATMI server thread |
| tpsvrthrinit(3c) | Initializes a BEA Tuxedo ATMI server thread |

**Table 1 BEA Tuxedo ATMI C Functions (Continued)**

| Name | Description |
|---|---|
| tpterm(3c) | Leaves an application |
| tptypes(3c) | Routine to determine information about a typed buffer |
| tpunadvertise(3c) | Routine for unadvertising a service name |
| tpunsubscribe(3c) | Unsubscribes to an event |
| tputrace(3c) | User-defined routine to provide trace information |
| tpxmltofml32(3c) | Converts XML buffer data to FML32 buffer data |
| tpxmltofml(3c) | Converts XML buffer data to FML buffer data |
| TRY(3c) | Exception-returning interface |
| tuxgetenv(3c) | Returns value for environment name |
| tuxgetmbaconv(3c) | Gets the value for environment variable TPMBACONV in the process environment |
| tuxgetmbenc(3c) | Gets the code-set encoding name for environment variable TPMBENC in the process environment |
| tuxputenv(3c) | Changes or adds value to environment |
| tuxreadenv(3c) | Adds variables to the environment from a file |
| tuxsetmbaconv(3c) | Sets the value for environment variable TPMBACONV in the process environment |
| tuxsetmbenc(3c) | Sets the code-set encoding name for environment variable TPMBENC in the process environment |
| tx_begin(3c) | Begins a global transaction |
| tx_close(3c) | Closes a set of resource managers |
| tx_commit(3c) | Commits a global transaction |
| tx_info(3c) | Returns global transaction information |
| tx_open(3c) | Opens a set of resource managers |

**Table 1 BEA Tuxedo ATMI C Functions (Continued)**

| Name | Description |
|---|---|
| tx_rollback(3c) | Rolls back a global transaction |
| tx_set_commit_return(3c) | Sets *commit_return* characteristic |
| tx_set_transaction_control(3c) | Sets *transaction_control* characteristic |
| tx_set_transaction_timeout(3c) | Sets *transaction_timeout* characteristic |
| userlog(3c) | Writes a message to the BEA Tuxedo ATMI system central event log |
| Usignal(3c) | Signal handling in a BEA Tuxedo ATMI system environment |
| Uunix_err(3c) | Prints UNIX system call error |

# Introduction to the C Language Application-to-Transaction Monitor Interface

## Description

The Application-to-Transaction Monitor Interface (ATMI) provides the interface between the application and the transaction processing system. This interface is known as the ATMI interface. It provides function calls to open and close resources, manage transactions, manage typed buffers, and invoke request/response and conversational service calls.

## Communication Paradigms

The function calls described in the ATMI reference pages imply a particular model of communication. This model is expressed in terms of how client and server processes can communicate using request and reply messages.

There are two basic communication paradigms: request/response and conversational. Request/response services are invoked by service requests along with their associated data. Request/response services can receive exactly one request (upon entering the service routine) and send at most one reply (upon returning from the service routine). Conversational services, on the other hand, are invoked by connection requests along with a means of referring to the open connection (that is, a descriptor used in calling subsequent connection routines). Once the connection has been established and the service routine invoked, either the connecting program or the conversational service can send and receive data as defined by the application until the connection is torn down.

Note that a process can initiate both request/response and conversational communication, but cannot accept both request/response and conversational service requests. The following sections describe the two communication paradigms in greater detail.

**Note:** In various parts of the BEA Tuxedo documentation we refer to *threads*. When this term is used in a discussion of multithreaded applications, it is self-explanatory. In some instances, however, the term is used in a discussion of a topic that is relevant for both single-threaded and multithreaded applications. In such cases, readers who are running single-threaded applications may assume that the term *thread* refers to an entire process.

## BEA Tuxedo ATMI System Request/ Response Paradigm for Client/Server

With regard to request/response communication, a client is defined as a process that can send requests and receive replies. By definition, clients cannot receive requests nor send replies. A client can send any number of requests, and can wait for the replies synchronously or receive

(some limited number of) the replies at its convenience. In certain cases, a client can send a request that has no reply. `tpinit()` and `tpterm()` allow a client to join and leave a BEA Tuxedo ATMI system application.

A request/response server is a process that can receive one (and only one) service request at a time and send at most one reply to that request. (If the server is multithreaded, however, it can receive multiple requests at one time and issue multiple replies at one time.) While a server is working on a particular request, it can act like a client by initiating request/response or conversational requests and receiving their replies. In such a capacity, a server is called a requester. Note that both client and server processes can be requesters (in fact, a client can be nothing but a requester).

A request/response server can forward a request to another request/response server. Here, the server passes along the request it received to another server and does not expect a reply. It is the responsibility of the last server in the chain to send the reply to the original requester. Use of the forwarding routine ensures that the original requester ultimately receives its reply.

Servers and service routines offer a structured approach to writing BEA Tuxedo ATMI system applications. In a server, the application writer can concentrate on the work performed by the service rather than communications details such as receiving requests and sending replies. Because many of the communication details are handled by BEA Tuxedo ATMI system's `main`, the application must adhere to certain conventions when writing a service routine. At the time a server finishes its service routine, it can send a reply using `tpreturn()` or forward the request using `tpforward()`. A service is not allowed to perform any other work nor is it allowed to communicate with any other process after this point. Thus, a service performed by a server is started when a request is received and ended when either a reply is sent or the request is forwarded.

Concerning request and reply messages, there is an inherent difference between the two: a request has no associated context before it is sent, but a reply does. For example, when sending a request, the caller must supply addressing information, whereas a reply is always returned to the process that originated the request, that is, addressing context is maintained for a reply and the sender of the reply can exert no control over its destination. The differences between the two message types manifest themselves in the parameters and descriptions of the routines described in `tpcall()`.

When a request message is sent, it is sent at a particular priority. The priority affects how a request is dequeued: when a server dequeues requests, it dequeues the one with the highest priority. To prevent starvation, the oldest request is dequeued every so often regardless of priority. By default, a request's priority is associated with the service name to which the request is being sent. Service names can be given priorities at configuration time (see `UBBCONFIG(5)`). A default priority is used if none is defined. In addition, the priority can be set at run time using a routine, `tpsprio()`. By doing so, the caller can override the configuration or default priority when the message is sent.

## BEA Tuxedo ATMI System Conversational Paradigm for Client/Server

With regard to conversational communication, a client is defined as a process that can initiate a conversation but cannot accept a connection request.

A conversational server is a process that can receive connection requests. Once the connection has been established and the service routine invoked, either the connecting program or the conversational service can send and receive data as defined by the application until the connection is torn down. The conversation is half-duplex in nature such that one side of the connection has control and can send data until it gives up control to the other side. In a single-threaded server, while the connection is established, the server is "reserved" such that no other process can establish a connection with it. When a connection is established to a multithreaded server, however, that server is not reserved for exclusive use by one process. Instead, it can accept requests from multiple client threads.

As with a request/response server, the conversational server can act as a requester by initiating other requests or connections with other servers. Unlike a request/response server, a conversational server cannot forward a request to another server. Thus, a conversational service performed by a server is started when a request is received and ended when the final reply is sent via `tpreturn()`.

Once the connection is established, the connection descriptor implies any context needed regarding addressing information for the participants. Messages can be sent and received as needed by the application. There is no inherent difference between the request and reply messages and no notion of priority of messages.

## Message Delivery

Sending and receiving messages, whether in conversation mode or request/response mode, implies communication between two units of an application. The great majority of messages lead to a reply or at least an acknowledgment, so that is an assurance that the message was received. There are, however, certain messages (some originated by the system, others originated by an application) where a reply or acknowledgment is not expected. For example, the system can send an unsolicited message using `tpnotify()` without the `TPACK()` flag, or an application can send a message using `tpacall()` with the `TPNOREPLY()` flag. If the message queue of the receiving program is full, the message is dropped.

If the sending and receiving side are on different machines, the communication takes place between bridge processes that send and receive messages across a network. This raises the additional possibility of non-delivery due to a circuit failure. Even when either of these conditions leads to the positing of an event or to a `ULOG` message, it is not easy to associate the event or `ULOG` message with the non-arrival of a particular message.

Because the BEA Tuxedo ATMI system is designed to handle large volumes of messages across broad networks, it is not programmed to detect and correct the small percentage of failures-to-deliver described in the preceding paragraphs. For that reason, there can be no guarantee that every message will be delivered.

## Message Sequencing

In the conversational model, for messages being exchanged using `tpsend()` and `tprecv()`, a sequence number is added to the message header and messages are received in the order in which they are sent. If a server or client gets a message out of order, the conversation is stopped, any transaction in progress is rolled back, and message `1572` in `LIBTUX`, "`Bad Conversational Sequence Number`," is logged.

In the Request/Response model, messages are not sequenced by the system. If the application logic implies a sequence, it is the responsibility of the application to monitor and control it. The parallel message transmission made possible by the support of multiple network addresses for bridge processes increases the possibility that messages will not be received in the order sent. An application that is concerned about this may choose to specify a single network address for each bridge process, add sequence numbers to their messages or require periodic acknowledgments.

## Queued Message Model

The BEA Tuxedo ATMI system queued message model allows for enqueuing a request message to stable storage for subsequent processing without waiting for its completion, and optionally getting a reply via a queued response message. The ATMI functions that queue messages and dequeue responses are `tpenqueue()` and `tpdequeue()`. They can be called from any type of BEA Tuxedo ATMI system application processes: client, server, or conversational. The functions `tpenqueue()` and `tpdequeue()` can also be used for peer-to-peer communication where neither the enqueuing application nor the dequeuing application are designated as server or client.

The queued message facility is an XA-compliant resource manager. Persistent messages are enqueued and dequeued within transactions to ensure one-time-only processing.

## ATMI Transactions

The BEA Tuxedo ATMI system supports two sets of mutually exclusive functions for defining and managing transactions: the BEA Tuxedo system's ATMI transaction demarcation functions (the names of which include the prefix `tp`) and X/Open's TX Interface functions (the names of which include the prefix `tx_`). Because X/Open used ATMI's transaction demarcation functions as the base for the TX Interface, the syntax and semantics of the TX Interface are quite similar to those of the ATMI. This section is an overview of ATMI transaction concepts. The next section introduces additional concepts about the TX Interface.

In the BEA Tuxedo ATMI system, a *transaction* is used to define a single logical unit of work that either wholly succeeds or has no effect whatsoever. A transaction allows work performed in many processes, possibly at different sites, to be treated as an atomic unit of work. The initiator of a transaction normally uses `tpbegin()` and either `tpcommit()` or `tpabort()` to delineate the operations within a transaction.

The initiator may also suspend its work on the current transaction by issuing `tpsuspend()`. Another process may take over the role of the initiator of a suspended transaction by issuing `tpresume()`. As a transaction initiator, a process must call one of the following: `tpsuspend()`, `tpcommit()`, or `tpabort()`. Thus, one process can start a transaction that another may finish.

If a process calling a service is in transaction mode, then the called service routine is also placed in transaction mode on behalf of the same transaction. Otherwise, whether the service is invoked in transaction mode or not depends on options specified for the service in the configuration file. A service that is not invoked in transaction mode can define multiple transactions between the time it is invoked and the time it ends. On the other hand, a service routine invoked in transaction mode can participate in only one transaction, and work on that transaction is completed upon termination of the service routine. Note that a connection cannot be upgraded to transaction mode: if `tpbegin()` is called while a conversation exists, the conversation remains outside of the transaction (as if `tpconnect()` had been called with the `TPNOTRAN()` flag).

A service routine joining a transaction that was started by another process is called a *participant*. A transaction can have several participants. A service can be invoked to do work on the same transaction more than once. Only the initiator of a transaction (that is, a process calling either `tpbegin()` or `tpresume()`) can call `tpcommit()` or `tpabort()`. Participants influence the outcome of a transaction by using `tpreturn()` or `tpforward()`. These two calls signify the end of a service routine and indicate that the routine has finished its part of the transaction.

## TX Transactions

Transactions defined by the TX Interface are practically identical with those defined by the ATMI functions. An application developer may use either set of functions when writing clients and service routines, but should not intermingle one set of functions with the other within a single process (that is, a process cannot call `tpbegin()` and later call `tx_commit()`).

The TX Interface has two calls for opening and closing resource managers in a portable manner, `tx_open()` and `tx_close()`, respectively. Transactions are started with `tx_begin()` and completed with either `tx_commit()` or `tx_rollback()`. `tx_info()` is used to retrieve transaction information, and there are three calls to set options for transactions: `tx_set_commit_return()`, `tx_set_transaction_control()`, and

`tx_set_transaction_timeout()`. The TX Interface has no equivalents to ATMI's `tpsuspend()` and `tpresume()`.

In addition to the semantics and rules defined for ATMI transactions, the TX Interface has some additional semantics that are worth introducing here. First, service routine writers wanting to use the TX Interface must supply their own `tpsvrinit()` routine that calls `tx_open()`. The default BEA Tuxedo ATMI system-supplied `tpsvrinit()` calls `tpopen()`. The same rule applies for `tpsvrdone()`: if the TX Interface is being used, then service routine writers must supply their own `tpsvrdone()` that calls `tx_close()`.

Second, the TX Interface has two additional semantics not found in ATMI. These are chained and unchained transactions, and transaction characteristics.

## Chained and Unchained Transactions

The TX Interface supports chained and unchained modes of transaction execution. By default, clients and service routines execute in the unchained mode; when an active transaction is completed, a new transaction does not begin until `tx_begin()` is called.

In the chained mode, a new transaction starts implicitly when the current transaction completes. That is, when `tx_commit()` or `tx_rollback()` is called, the BEA Tuxedo ATMI system coordinates the completion of the current transaction and initiates a new transaction before returning control to the caller. (Certain failure conditions may prevent a new transaction from starting.)

Clients and service routines enable or disable the chained mode by calling `tx_set_transaction_control()`. Transitions between the chained and unchained mode affect the behavior of the next `tx_commit()` or `tx_rollback()` call. The call to `tx_set_transaction_control()` does not put the caller into or take it out of transaction mode.

Since `tx_close()` cannot be called when the caller is in transaction mode, a caller executing in chained mode must switch to unchained mode and complete the current transaction before calling `tx_close()`.

## Transaction Characteristics

A client or a service routine may call `tx_info()` to obtain the current values of their transaction characteristics and to determine whether they are executing in transaction mode.

The state of an application process includes several transaction characteristics. The caller specifies these by calling `tx_set_*()` functions. When a client or a service routine sets the value of a characteristic, it remains in effect until the caller specifies a different value. When the caller obtains the value of a characteristic via `tx_info()`, it does not change the value.

## Error Handling

Most of the ATMI functions have one or more error returns. An error condition is indicated by an otherwise impossible returned value. This is usually -1 or error, or 0 for a bad field identifier (BADFLDID) or address. The error type is also made available in the external integer tperrno. tperrno is not cleared on successful calls, so it should be tested only after an error has been indicated.

The tpstrerror() function is provided to produce a message on the standard error output. It takes one argument, an integer (found in tperrno) and returns a pointer to the text of an error message in LIBTUX_CAT. The pointer can be used as an argument to userlog().

tperrordetail() can be used as the first step of a three step procedure to get additional detail about an error in the most recent BEA Tuxedo ATMI system call on the current thread. tperrordetail() returns an integer which is then used as an argument to tpstrerrordetail() to retrieve a pointer to a string that contains the error message. The pointer can then be used as an argument to userlog or to fprintf().

The error codes that can be produced by an ATMI function are described on each ATMI reference page. The F_error() and F_error32() functions are provided to produce a message on the standard error output for FML errors. They take one parameter, a string; print the argument string appended with a colon and a blank; and then print an error message followed by a newline character. The error message displayed is the one defined for the error number currently in Ferror() or Ferror32(), which is set when errors occur.

Fstrerror(), and its counterpart, Fstrerror32(), can be used to retrieve the text of an FML error message from a message catalog; it returns a pointer that can be used as an argument to userlog.

The error codes that can be produced by an FML function are described on each FML reference page.

## Timeouts

There are three types of timeouts in the BEA Tuxedo ATMI system: one is associated with the duration of a transaction from start to finish. A second is associated with the maximum length of time a blocking call will remain blocked before the caller regains control. The third is a service timeout and occurs when a call exceeds the number of seconds specified in the SVCTIMEOUT parameter in the SERVICES section of the configuration file.

The first kind of timeout is specified when a transaction is started with tpbegin(). (See tpbegin(3c) for details.) The second kind of timeout can occur when using the BEA Tuxedo ATMI system communication routines defined in tpcall(3c). Callers of these routines

typically block when awaiting a reply that has yet to arrive, although they can also block trying to send data (for example, if request queues are full). The maximum amount of time a caller remains blocked is determined by a BEA Tuxedo ATMI system configuration file parameter. (See the BLOCKTIME parameter in UBBCONFIG(5) for details.)

Blocking timeouts are performed by default when the caller is not in transaction mode. When a client or server is in transaction mode, it is subject to the timeout value with which the transaction was started and is not subject to the blocking timeout value specified in the UBBCONFIG file.

When a transaction timeout occurs, replies to asynchronous requests made in transaction mode become invalid. That is, if a process is waiting for a particular asynchronous reply for a request sent in transaction mode and a transaction timeout occurs, the descriptor for that reply becomes invalid. Similarly, if a transaction timeout occurs, an event is generated on the connection descriptor associated with the transaction and that descriptor becomes invalid. On the other hand, if a blocking timeout occurs, the descriptor is still valid and the waiting process can reissue the call to await the reply.

The service timeout mechanism provides a way for the system to kill processes that may be frozen by some unknown or unexpected system error. When a service timeout occurs in a request/response service, the BEA Tuxedo ATMI system kills the server process that is executing the frozen service and returns error code TPESVCERR. If a service timeout occurs in a conversational service, the TP_EVSVCERR event is returned.

If a transaction has timed out, the only valid communications before the transaction is aborted are calls to tpacall() with TPNOREPLY, TPNOTRAN, and TPNOBLOCK set.

Since release 6.4, some additional detail has been provided beyond the TPESVCERR error code. If a service fails due to exceeding the timeout threshold, an event, .SysServiceTimeout, is posted.

## Dynamic Service Advertisements

By default, a server's services are advertised when it is booted and unadvertised when it is shut down. If a server needs to control the set of services that it offers at run time, it can do so by calling tpadvertise() and tpunadvertise(). These routines affect only the services offered by the calling server unless that server belongs to a Multiple Server, Single Queue (MSSQ) set. Because all servers in an MSSQ set must offer the same set of services, these routines also affect the advertisements of all servers sharing the caller's MSSQ set.

## Buffer Management

Initially, a process has no buffers. Before sending a message, a buffer must be allocated using tpalloc(). The sender's data can then be placed in the buffer and sent. This buffer has a specific

structure. The particular structure is denoted by the *type* argument to the `tpalloc()` function. Since some structures can need further classification, a subtype can also be given (for example, a particular type of C structure).

When receiving a message, a buffer is required into which application data can be received. This buffer must be one originally gotten from `tpalloc()`. Note that a BEA Tuxedo ATMI system server, in its `main`, allocates a buffer whose address is passed to a request/response or conversational service upon invoking the service. (See `tpservice(3c)` for details on how this buffer is treated.)

Buffers used for receiving messages are treated slightly differently than those used for sending: the size and address usually change upon receipt of a message, since the system internally swaps the buffer passed into the receive call with internal buffers it used to process the buffer. A buffer may grow or shrink when it receives data. Whether it grows or shrinks depends on the amount of data sent by the sender, and the internal data flow needed to get the data from sender to receiver. Many factors can affect the buffer size, including compression, receiving a message from a different type of machine, and the action of the `postrecv()` function for the type of buffer being used (see `buffer(3c)`). The buffer sizes in Workstation clients are usually different from those in native clients.

It is best to think of the receive buffer as a placeholder, rather than the actual container that will receive the message. The system sometimes uses the size of the buffer you pass as a hint, so it does help if it is big enough to hold the expected reply.

On the sending side, buffer types that might be filled to less than their allocated capacity (for example, FML or STRING buffers) send only the amount used. A 100K FML32 buffer with one integer field in it is sent as a much smaller buffer, containing only that integer.

This means that the receiver will receive a buffer smaller than what was originally allocated by the sender, yet larger than the data that was sent. For example, if a STRING buffer of 10K bytes is allocated, and the string “HELLO” is copied into it, only the six bytes are sent, and the receiver will probably end up with a buffer that is around 1K or 4K bytes. (It may be larger or smaller, depending on other factors.) The BEA Tuxedo ATMI system guarantees only that a received message will contain all of the data that was sent; it does not guarantee that the message will contain all the free space it originally contained.

The process receiving the reply is responsible for noting size changes in the buffer (using `tptypes()`) and reallocating the buffer if necessary. All BEA Tuxedo ATMI functions change a receiver’s buffer return information about the amount of data in the buffer, so it should become standard practice to check the buffer size every time a reply is received.

One can send and receive messages using the same data buffer. Alternatively, a different data buffer can be allocated for each message. It is usually the responsibility of the calling process to free its buffers by invoking `tpfree()`. However, in limited cases, the BEA Tuxedo ATMI system frees the caller's buffer. For more information about buffer usage, see the descriptions of communication functions such as `tpfree()`.

## Buffer Type Switch

The `tmtype_sw_t` structure provides the description required when adding new buffer types to `tm_typesw()`, the buffer type switch for a process. The switch elements are defined in `typesw(5)`. The function names used in this entry are templates for the actual function names defined by the BEA Tuxedo ATMI system or by applications in which custom buffer types are created. These function names can be mapped easily to switch elements: to create a template name simply add the prefix `_tm` to the element name of a function pointer. For example, the template name for the element `initbuf` is `_tminitbuf`.

The `type` element must be non-NULL and at most 8 characters in length. If this element is not unique in the switch, then `subtype()` must be non-NULL.

The `subtype()` element can be NULL, a string of at most 16 characters, or * (the wildcard character). The combination of `type()` and `subtype()` must uniquely identify an element in the switch.

A given type can have multiple subtypes. If all subtypes are to be treated the same for a given type, then the wildcard character, "*", can be used. Note that the `tptypes()` function can be used to determine a buffer's type and subtype if subtypes need to be distinguished. If some subset of the subtypes within a particular type are to be treated individually, and the rest are to be treated identically, then those that are to be singled out with specific subtype values should appear in the switch before the subtype designated with the wildcard. Thus, searching for types and subtypes in the switch is done from top to bottom, and the wildcard subtype entry accepts any "leftover" type matches.

The `dfltsize()` element is used when allocating or reallocating a buffer. The semantics of `tpalloc()` and `tprealloc()` are such that the larger of the following two values is used to create or reallocate a buffer: the value of `dfltsize()` or the value of the *`size`* parameter for the `tpalloc()` and `tprealloc()` functions. For some types of structures, such as a fixed-sized C structure, the buffer size should equal the size of the structure. If `dfltsize()` is set to this value, then the caller may not need to specify the buffer's length to routines in which a buffer is passed. `dfltsize()` can be 0 or less. However, if `tpalloc()` or `tprealloc()` is called and the *`size`* parameter for the function being called is also less than or equal to 0, then the routine will fail. We recommend setting `dfltsize()` to a value greater than 0.

The BEA Tuxedo ATMI system provides five basic buffer types:

- CARRAY—a character array, possibly containing NULL characters, which is neither encoded nor decoded during transmission
- STRING—a NULL-terminated character array
- FML—fielded buffers (FML or FML32)
- XML—XML document or datagram buffer
- VIEW—simple C structures (VIEW or VIEW32); all views are handled by the same set of routines. The name of a particular view is its subtype name.

Two of these buffer types have synonyms: X_OCTET is a synonym for CARRAY, and both X_C_TYPE and X_COMMON are synonyms for VIEW. X_C_TYPE supports all the same elements as VIEW, whereas X_COMMON supports only longs, shorts, and characters. X_COMMON should be used when both C and COBOL programs are communicating.

An application wishing to supply its own buffer type can do so by adding an instance to the tm_typesw() array. Whenever adding or deleting a buffer type, be careful to leave a NULL entry at the end of the array. Note that a buffer type with a NULL name is not permitted. An application client or server is linked with the new buffer type switch by explicitly specifying the name of the source or object file on the buildserver() or buildclient() command line using the -f option.

## Unsolicited Notification

There are two methods for sending messages to application clients outside the boundaries of the client/server interaction defined above. The first is the broadcast mechanism supported by tpbroadcast(). This function allows application clients, servers, and administrators to broadcast typed buffer messages to a set of clients selected on the basis of the names assigned to them. The names assigned to clients are determined in part by the application (specifically, by the information passed in the TPINIT typed buffer at tpinit() time) and in part by the system (based on the processor through which the client accesses the application).

The second method is the notification of a particular client as identified from an earlier or current service request. Each service request contains a unique client identifier that identifies the originating client for the service request. Calls to the tpcall() and tpforward() functions from within a service routine do not change the originating client for that chain of service requests. Client identifiers can be saved and passed between application servers. The tpnotify() function is used to notify clients identified in this manner.

## Single or Multiple Application Contexts per Process

The BEA Tuxedo ATMI system allows client programs to create an association with one or more applications per process. If `tpinit()` is called with the `TPMULTICONTEXTS` parameter included in the *`flags`* field of the `TPINIT` structure, then multiple client contexts are allowed. If `tpinit()` is called implicitly, is called with a NULL parameter, or the *`flags`* field does not include `TPMULTICONTEXTS`, then only a single application association is allowed.

In single-context mode, if `tpinit()` is called more than once (that is, if it is called after the client has already joined the application), no action is taken and success is returned.

In multicontext mode, each call to `tpinit()` creates a new application association. The application can obtain a handle representing this application association by calling `tpgetctxt()`. Any thread in the same process can call `tpsetctxt()` to set that thread's context.

Once an application has chosen single-context mode, all calls to `tpinit()` must specify single-context mode until all application associations are terminated. Similarly, once an application has chosen multicontext mode, all calls to `tpinit()` must specify multicontext mode until all application associations are terminated.

Server programs can be associated with only a single application and cannot act as clients. However, within each server program, there may be multiple server dispatch contexts. Each server dispatch context works in its own thread.

Table 2 shows the transitions that may occur, within a client process, among the following states: the uninitialized state, the initialized in single-context mode state, and the initialized in multicontext mode state.

**Table 2 Per-Process Context Modes**

| Function | States | | |
|---|---|---|---|
| | **Uninitialized $S_0$** | **Initialized Single-context Mode $S_1$** | **Initialized Multicontext Mode $S_2$** |
| `tpinit` without `TPMULTICONTEXTS` | $S_1$ | $S_1$ | $S_2$(*error*) |
| `tpinit` with `TPMULTICONTEXTS` | $S_2$ | $S_1$ (*error*) | $S_2$ |
| Implicit `tpinit` | $S_1$ | $S_1$ | $S_2$ (*error*) |

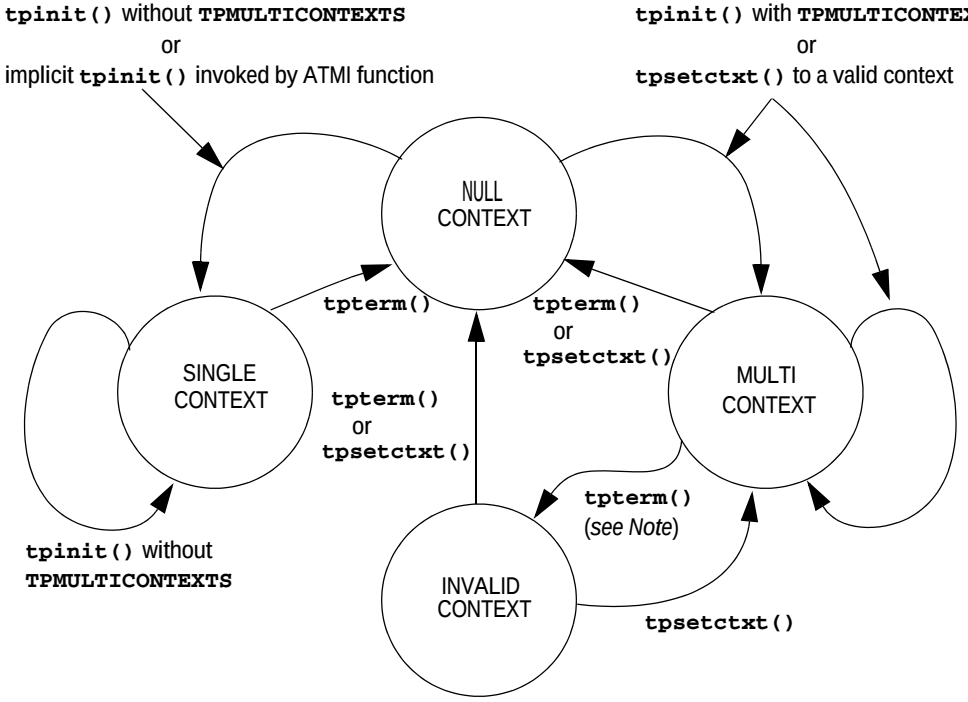
**Table 2 Per-Process Context Modes**

| Function | States | | |
|---|---|---|---|
| | Uninitialized $S_0$ | Initialized Single-context Mode $S_1$ | Initialized Multicontext Mode $S_2$ |
| `tpterm`—not last association | | | $S_2$ |
| `tpterm`—last association | | $S_0$ | $S_0$ |
| `tpterm`—no association | $S_0$ | | |

## Context State Changes for a Client Thread

In a multicontext application, calls to various functions result in context state changes for the calling thread and any other threads that are active in the same context as the calling process. The following diagram illustrates the context state changes that result from calls to the `tpinit()`, `tpsetctxt()`, and `tpterm()` functions. (The `tpgetctxt()` function does not produce any context state changes.)

**Multicontext State Transitions**




**Note:** When `tpterm()` is called by a thread running in the multicontext state (`TPMULTICONTEXTS`), the calling thread is placed in the NULL context state (`TPNULLCONTEXT`). All other threads associated with the terminated context are switched to the invalid context state (`TPINVALIDCONTEXT`).

Table 3 lists all possible context state changes produced by calling `tpinit()`, `tpsetctxt()`, and `tpterm()`. These states are thread-specific; different threads can be in different states when they are part of a multicontexted application. By contrast, each context state listed in the preceding table (“Per-Process Context Modes”) applies to an entire process.

**Table 3 Context State Changes for a Client Thread**

| When this function is executed . . . | Then a thread in this context state results in . . . | | | |
|---|---|---|---|---|
| | **NULL Context** | **Single Context** | **Multicontext** | **Invalid Context** |
| `tpinit` without `TPMULTICONTEXTS` | Single context | Single context | Error | Error |
| `tpinit` with `TPMULTICONTEXTS` | Multicontext | Error | Multicontext | Error |
| `tpsetctxt` to `TPNULLCONTEXT` | NULL | Error | NULL | NULL |
| `tpsetctxt` to context 0 | Error | Single context | Error | Error |
| `tpsetctxt` to context > 0 | Multicontext | Error | Multicontext | Multicontext |
| Implicit `tpinit` | Single context | N/A | N/A | Error |
| `tpterm` in this thread | NULL | NULL | NULL | NULL |
| `tpterm` in a different thread of this context | N/A | NULL | Invalid | N/A |

## Support for Threads Programming

The BEA Tuxedo ATMI system supports multithreaded programming in several ways. If the process is using single-context mode, then as the application creates new threads, those threads share the BEA Tuxedo ATMI context for the process. In a client, after a thread issues a `tpinit()` call in single-context mode, other threads may then proceed to issue ATMI calls. For example, one thread may issue a `tpacall()` and a different thread in the same process may issue a `tpgetrply()`.

When in multicontext mode, threads initially are not associated with a BEA Tuxedo ATMI application. A thread can either join an existing application association by calling `tpsetctxt()` or create a new association by calling `tpinit()` with the `TPMULTICONTEXTS` flag set.

Whether running in single-context mode or multicontext mode, the application is responsible for coordinating its threads so that ATMI operations are performed at the appropriate time.

An application may create additional threads within a server by using OS thread functions. These threads may operate independently of the BEA Tuxedo ATMI system, or they may operate in the same context as one of the server dispatch threads. Initially, application-created server threads are not associated with any server dispatch context. An application-created server thread may call `tpsetctxt()` to associate itself with a server dispatch thread. The application-created server thread must complete all of its ATMI calls before the dispatched thread calls `tpreturn()` or `tpforward()`. A server thread dispatched by the BEA Tuxedo ATMI system may not call `tpsetctxt()`. In addition, application-created threads may not make ATMI calls that would cause an implicit `tpinit()` when not associated with a context. On the other hand, this failure to make ATMI calls does not occur with dispatcher-created threads because those threads are always associated with a context. All server threads are prohibited from calling `tpinit()`.

In a multithreaded application, a thread that is operating in the `TPINVALIDCONTEXT` state is prohibited from calling many ATMI functions. The following lists specify which functions may and may not be called under these circumstances.

The BEA Tuxedo ATMI system allows a thread operating in the `TPINVALIDCONTEXT` state to call the following functions:

- `catgets(3c)`
- `catopen, catclose(3c)`
- `decimal(3c)`
- `gp_mktime(3c)`
- `nl_langinfo(3c)`
- `setlocale(3c)`
- `strerror(3c)`
- `strftime(3c)`
- `tpalloc(3c)`
- `tpconvert(3c)`
- `tpcryptpw(3c)`
- `tperrordetail(3c)`
- `tpfml32toxml(3c)`
- `tpfmltoxml(3c)`
- `tpfree(3c)`
- `tpgblktime(3c)`

- tpgetctxt(3c)
- tpgetrepos(3c)
- tprealloc(3c)
- tpsblktime(3c)
- tpsetctxt(3c)
- tpsetrepos(3c)
- tpstrerror(3c)
- tpstrerrordetail(3c)
- tpterm(3c)
- tptypes(3c)
- tpxmltofml32(3c)
- tpxmltofml(3c)
- TRY(3c)
- tuxgetenv(3c)
- tuxputenv(3c)
- tuxreadenv(3c)
- userlog(3c)
- Usignal(3c)
- Uunix_err(3c)

The BEA Tuxedo ATMI system *does not allow* a thread operating in the TPINVALIDCONTEXT state to call the following functions:

- AEWsetunsol(3c)
- tpabort(3c)
- tpacall(3c)
- tpadmcall(3c)
- tpbegin(3c)
- tpbroadcast(3c)
- tpcall(3c)
- tpcancel(3c)
- tpchkauth(3c)

- tpchkunsol(3c)
- tpclose(3c)
- tpcommit(3c)
- tpconnect(3c)
- tpdequeue(3c)
- tpenqueue(3c)
- tpgetadmkey(3c)
- tpgetlev(3c)
- tpgetrply(3c)
- tpgprio(3c)
- tpinit(3c)
- tpnotify(3c)
- tpopen(3c)
- tppost(3c)
- tprecv(3c)
- tpresume(3c)
- tpscmt(3c)
- tpsend(3c)
- tpsetunsol(3c)
- tpsprio(3c)
- tpsubscribe(3c)
- tpsuspend(3c)
- tpunsubscribe(3c)
- tx_begin(3c)
- tx_close(3c)
- tx_commit(3c)
- tx_info(3c)
- tx_open(3c)
- tx_rollback(3c)
- tx_set_commit_return(3c)

- tx_set_transaction_control(3c)
- tx_set_transaction_timeout(3c)

## C Language ATMI Return Codes and Other Definitions

The following return code and flag definitions are used by the ATMI routines. For an application to work with different transaction monitors without change or recompilation, each system must define its flags and return codes as follows:

```
/*
 * The following definitions must be included in atmi.h
 */



/* Flags to service routines */

 #define TPNOBLOCK     0x00000001 /* non-blocking send/rcv */
 #define TPSIGRSTRT    0x00000002 /* restart rcv on interrupt */
 #define TPNOREPLY     0x00000004 /* no reply expected */
 #define TPNOTRAN      0x00000008 /* not sent in transaction mode */
 #define TPTRAN        0x00000010 /* sent in transaction mode */
 #define TPNOTIME      0x00000020 /* no timeout */
 #define TPABSOLUTE    0x00000040 /* absolute value on tmsetprio */
 #define TPGETANY      0x00000080 /* get any valid reply */
 #define TPNOCHANGE    0x00000100 /* force incoming buffer to match */
 #define RESERVED_BIT1 0x00000200 /* reserved for future use */
 #define TPCONV        0x00000400 /* conversational service */
 #define TPSENDONLY    0x00000800 /* send-only mode */
 #define TPRECVONLY    0x00001000 /* recv-only mode */
 #define TPACK         0x00002000 /* */

/* Flags to tpreturn - also defined in xa.h */
 #define TPFAIL        0x20000000 /* service FAILURE for tpreturn */
 #define TPEXIT        0x08000000 /* service FAILURE with server exit */
 #define TPSUCCESS     0x04000000 /* service SUCCESS for tpreturn */

/* Flags to tpscmt - Valid TP_COMMIT_CONTROL
 * characteristic values
 */
 #define TP_CMT_LOGGED 0x01       /* return after commit
                                   * decision is logged */
 #define TP_CMT_COMPLETE 0x02     /* return after commit has
                                   * completed */



/* client identifier structure */
```

```
 struct clientid_t {
 long clientdata[4];              /* reserved for internal use */
 }
 typedef struct clientid_t CLIENTID;
 /* context identifier structure */
 typedef long TPCONTEXT_T;
 /* interface to service routines */
 struct tpsvcinfo {
 name[32];
 long flags;                      /* describes service attributes */
 char *data;                      /* pointer to data */
 long len;                        /* request data length */
 int cd;                          /* connection descriptor
 * if (flags  TPCONV) true */
 long appkey;                     /* application authentication client
 * key */
 CLIENTID cltid;                  /* client identifier for originating
  * client */
 };

typedef struct tpsvcinfo TPSVCINFO;

/* tpinit(3c) interface structure */
 #define MAXTIDENT                30

struct tpinfo_t {
 char usrname[MAXTIDENT+2];       /* client user name */
 char cltname[MAXTIDENT+2];       /* app client name */
 char passwd[MAXTIDENT+2];        /* application password */
 long flags;                      /* initialization flags */
 long datalen;                    /* length of app specific data */
 long data;                       /* placeholder for app data */
 };
 typedef struct tpinfo_t TPINIT;

/* The transactionID structure passed to tpsuspend(3c) and tpresume(3c) */
 struct tp_tranid_t {
 long info[6];                    /* Internally defined */
 };

typedef struct tp_tranid_t TPTRANID;

/* Flags for TPINIT */
 #define TPU_MASK                 0x00000007     /* unsolicited notification
                                                  * mask */
 #define TPU_SIG                  0x00000001     /* signal based
                                  * notification */
 #define TPU_DIP                  0x00000002     /* dip-in based
                                  * notification */
```

```
 #define TPU_IGN                  0x00000004     /* ignore unsolicited
                                                  * messages */
 #define TPU_THREAD               0x00000040     /* THREAD notification */
 #define TPSA_FASTPATH            0x00000008     /* System access ==
                                                  * fastpath */
 #define TPSA_PROTECTED           0x00000010     /* System access ==
                                                  * protected */
 #define TPMULTICONTEXTS          0x00000020     /* multiple context associa-
                                                  * tions per process */
 /*  /Q tpqctl_t data structure                  */
 #define TMQNAMELEN               15
 #define TMMSGIDLEN               32
 #define TMCORRIDLEN              32

struct tpqctl_t {                /* control parameters to queue primitives */
 long flags;                     /* indicates which values are set */
 long deq_time;                  /* absolute/relative time for dequeuing */
 long priority;                  /* enqueue priority */
 long diagnostic;                /* indicates reason for failure */
 char msgid[TMMSGIDLEN];         /* ID of message before which to queue */
 char corrid[TMCORRIDLEN];       /* correlation ID used to identify message */
 char replyqueue[TMQNAMELEN+1];  /* queue name for reply message */
 char failurequeue[TMQNAMELEN+1]; /* queue name for failure message */
 CLIENTID cltid;                 /* client identifier for */
                                 /* originating client */
 long urcode;                    /* application user-return code */
 long appkey;                    /* application authentication client key */
 long delivery_qos;              /* delivery quality of service */
 long reply_qos;                 /* reply message quality of service */
 long exp_time                   /* expiration time */
 };
 typedef struct tpqctl_t TPQCTL;

/* /Q structure elements that are valid - set in flags */
 #ifndef TPNOFLAGS
 #define TPNOFLAGS               0x00000    /* no flags set -- no get */
 #endif
 #define TPQCORRID               0x00001    /* set/get correlation ID */
 #define TPQFAILUREQ             0x00002    /* set/get failure queue */
 #define TPQBEFOREMSGID          0x00004    /* enqueue before message ID */
 #define TPQGETBYMSGIDOLD        0x00008    /* deprecated */
 #define TPQMSGID               0x00010    /* get msgid of enq/deq message */
 #define TPQPRIORITY             0x00020    /* set/get message priority */
 #define TPQTOP                  0x00040    /* enqueue at queue top */
 #define TPQWAIT                 0x00080    /* wait for dequeuing */
 #define TPQREPLYQ               0x00100    /* set/get reply queue */
 #define TPQTIME_ABS             0x00200    /* set absolute time */
 #define TPQTIME_REL             0x00400    /* set relative time */
 #define TPQGETBYCORRIDOLD       0x00800    /* deprecated */
```

```
 #define TPQPEEK                  0x01000    /* non-destructive dequeue */
 #define TPQDELIVERYQOS           0x02000    /* delivery quality of service */
 #define TPQREPLYQOS              0x04000    /* reply msg quality of service*/
 #define TPQEXPTIME_ABS           0x08000    /* absolute expiration time */
 #define TPQEXPTIME_REL           0x10000    /* relative expiration time */
 #define TPQEXPTIME_NONE          0x20000    /* never expire */
 #define TPQGETBYMSGID            0x40008    /* dequeue by msgid */
 #define TPQGETBYCORRID           0x80800    /* dequeue by corrid */

/* Valid flags for the quality of service fields in the TPQCTL structure */
 #define TPQQOSDEFAULTPERSIST     0x00001    /* queue's default persistence */
                                             /* policy */
 #define TPQQOSPERSISTENT         0x00002    /* disk message */
 #define TPQQOSNONPERSISTENT      0x00004    /* memory message */

/* error return codes */
 extern int tperrno;
 extern long tpurcode;

/* tperrno values - error codes */
 * The reference pages explain the context in which the following
 * error codes can return.
 */

 #define TPMINVAL                 0          /* minimum error message */
 #define TPEABORT                 1
 #define TPEBADDESC               2
 #define TPEBLOCK                 3
 #define TPEINVAL                 4
 #define TPELIMIT                 5
 #define TPENOENT                 6
 #define TPEOS                    7
 #define TPEPERM                  8
 #define TPEPROTO                 9
 #define TPESVCERR                10
 #define TPESVCFAIL               11
 #define TPESYSTEM                12
 #define TPETIME                  13
 #define TPETRAN                  14
 #define TPGOTSIG                 15
 #define TPERMERR                 16
 #define TPEITYPE                 17
 #define TPEOTYPE                 18
 #define TPERELEASE               19
 #define TPEHAZARD                20
 #define TPEHEURISTIC             21
 #define TPEEVENT                 22
 #define TPEMATCH                 23
 #define TPEDIAGNOSTIC            24
```

```
 #define TPEMIB                  25
 #define TPMAXVAL                26         /* maximum error message */

/* conversations - events */
 #define TPEV_DISCONIMM          0x0001
 #define TPEV_SVCERR             0x0002
 #define TPEV_SVCFAIL            0x0004
 #define TPEV_SVCSUCC            0x0008
 #define TPEV_SENDONLY           0x0020

/* /Q diagnostic codes          */
 #define QMEINVAL                -1
 #define QMEBADRMID              -2
 #define QMENOTOPEN              -3
 #define QMETRAN                 -4
 #define QMEBADMSGID             -5
 #define QMESYSTEM               -6
 #define QMEOS                   -7
 #define QMEABORTED              -8
 #define QMENOTA                 QMEABORTED
 #define QMEPROTO                -9
 #define QMEBADQUEUE             -10
 #define QMENOMSG                -11
 #define QMEINUSE                -12
 #define QMENOSPACE              -13
 #define QMERELEASE              -14
 #define QMEINVHANDLE            -15
 #define QMESHARE                -16

/* EventBroker Messages */
 #define TPEVSERVICE             0x00000001
 #define TPEVQUEUE               0x00000002
 #define TPEVTRAN                0x00000004
 #define TPEVPERSIST             0x00000008

/* Subscription Control Structure */
 struct tpevctl_t {
     long flags;
     char name1[XATMI_SERVICE_NAME_LENGTH];
     char name2[XATMI_SERVICE_NAME_LENGTH];
     TPQCTL qctl;
 };
 typedef struct tpevctl_t TPEVCTL;
```

## C Language TX Return Codes and Other Definitions

The following return code and flag definitions are used by the TX routines. For an application to work with different transaction monitors without change or recompilation, each system must define its flags and return codes as follows:

```
#define TX_H_VERSION           0           /* current version of this
                                            * header file */

 /*
  * Transaction identifier
  */
  #define XIDDATASIZE          128         /* size in bytes */
  struct xid_t {
         long formatID;                    /* format identifier */
         long gtrid_length;                /* value not to exceed 64 */
         long bqual_length;                /* value not to exceed 64 */
         char data[XIDDATASIZE];
  };
  typedef struct xid_t XID;
  /*
  * A value of -1 in formatID means that the XID is null.
  */

 /*
  * Definitions for tx_ routines
  */
  /* commit return values */
  typedef long COMMIT_RETURN;
  #define TX_COMMIT_COMPLETED 0
  #define TX_COMMIT_DECISION_LOGGED 1

 /* transaction control values */
  typedef long TRANSACTION_CONTROL;
  #define TX_UNCHAINED 0
  #define TX_CHAINED 1

 /* type of transaction timeouts */
  typedef long TRANSACTION_TIMEOUT;

 /* transaction state values */
  typedef long TRANSACTION_STATE;
  #define TX_ACTIVE 0
  #define TX_TIMEOUT_ROLLBACK_ONLY 1
  #define TX_ROLLBACK_ONLY 2

 /* structure populated by tx_info */
  struct tx_info_t {
         XID xid;
         COMMIT_RETURN when_return;
         TRANSACTION_CONTROL transaction_control;
         TRANSACTION_TIMEOUT transaction_timeout;
         TRANSACTION_STATE   transaction_state;
  };
  typedef struct tx_info_t TXINFO;
```

```
/*
 * tx_ return codes
 * (transaction manager reports to application)
 */
 #define TX_NOT_SUPPORTED            1 /* option not supported */
 #define TX_OK                       0 /* normal execution */
 #define TX_OUTSIDE                 -1 /* application is in an RM
                                        * local transaction */
 #define TX_ROLLBACK                -2 /* transaction was rolled
                                        * back */
 #define TX_MIXED                   -3 /* transaction was
                                        * partially committed and
                                        * partially rolled back */
 #define TX_HAZARD                  -4 /* transaction may have been
                                        * partially committed and
                                        * partially rolled back */
 #define TX_PROTOCOL_ERROR          -5 /* routine invoked in an
                                        * improper context */
 #define TX_ERROR                   -6 /* transient error */
 #define TX_FAIL                    -7 /* fatal error */
 #define TX_EINVAL                  -8 /* invalid arguments were given */
 #define TX_COMMITTED               -9 /* transaction has
                                        * heuristically committed */

 #define TX_NO_BEGIN                -100 /* transaction committed plus
                                          * new transaction could not
                                          * be started */
 #define TX_ROLLBACK_NO_BEGIN       (TX_ROLLBACK+TX_NO_BEGIN)
                                         /* transaction rollback plus
                                          * new transaction could not
                                          * be started */
 #define TX_MIXED_NO_BEGIN          (TX_MIXED+TX_NO_BEGIN)
                                         /* mixed plus new transaction
                                          * could not be started */
 #define TX_HAZARD_NO_BEGIN         (TX_HAZARD+TX_NO_BEGIN)
                                         /* hazard plus new transaction
                                          * could not be started */
 #define TX_COMMITTED_NO_BEGIN      (TX_COMMITTED+TX_NO_BEGIN)
                                         /* heuristically committed plus
                                          * new transaction could not
                                          * be started */
```

## ATMI State Transitions

The BEA Tuxedo ATMI system keeps track of the state for each process and verifies that legal state transitions occur for the various function calls and options. The state information includes the process type (request/response server, conversational server, or client), the initialization state

(uninitialized or initialized), the resource management state (closed or open), the transaction state of the process, and the state of all asynchronous request and connection descriptors. When an illegal state transition is attempted, the called function fails, setting `tperrno` to `TPEPROTO`. The legal states and transitions for this information are described in the following tables.

Table 4 indicates which functions may be called by request/response servers, conversational servers, and clients. Note that `tpsvrinit()`, `tpsvrdone()`, `tpsvrthrinit()`, and `tpsvrthrdone()` are not included in this table because they are not called by applications (that is, they are application-supplied functions that are invoked by the BEA Tuxedo ATMI system).

**Table 4 Available Functions**

| Function | Process Type | | |
|---|---|---|---|
| | Request/Response Server | Conversational Server | Client |
| tpabort | Y | Y | Y |
| tpacall | Y | Y | Y |
| tpadvertise | Y | Y | N |
| tpalloc | Y | Y | Y |
| tpbegin | Y | Y | Y |
| tpbroadcast | Y | Y | Y |
| tpcall | Y | Y | Y |
| tpcancel | Y | Y | Y |
| tpchkauth | Y | Y | Y |
| tpchkunsol | N | N | Y |
| tpclose | Y | Y | Y |
| tpcommit | Y | Y | Y |
| tpconnect | Y | Y | Y |
| tpdequeue | Y | Y | Y |
| tpdiscon | Y | Y | Y |

**Table 4 Available Functions (Continued)**

| Function | Process Type | | |
|---|---|---|---|
| | Request/Response Server | Conversational Server | Client |
| tpenqueue | Y | Y | Y |
| tpfml1toxml | Y | Y | Y |
| tpfml32toxml | Y | Y | Y |
| tpforward | Y | N | N |
| tpfree | Y | Y | Y |
| tpgblktime | Y | Y | Y |
| tpgetctxt | Y | Y | Y |
| tpgetlev | Y | Y | Y |
| tpgetrepos | Y | Y | N |
| tpgetrply | Y | Y | Y |
| tpgprio | Y | Y | Y |
| tpinit | N | N | Y |
| tpnotify | Y | Y | Y |
| tpopen | Y | Y | Y |
| tppost | Y | Y | Y |
| tprealloc | Y | Y | Y |
| tprecv | Y | Y | Y |
| tpresume | Y | Y | Y |
| tpreturn | Y | Y | N |
| tpsblktime | Y | Y | Y |
| tpscmt | Y | Y | Y |
| tpsend | Y | Y | Y |

**Table 4 Available Functions (Continued)**

| Function | Process Type | | |
|---|---|---|---|
| | Request/Response Server | Conversational Server | Client |
| tpservice | Y | Y | N |
| tpsetctxt | Y (*in application-created threads*) | Y (*in application-created threads*) | Y |
| tpsetrepos | Y | Y | N |
| tpsetunsol | N | N | Y |
| tpsprio | Y | Y | Y |
| tpsubscribe | Y | Y | Y |
| tpsuspend | Y | Y | Y |
| tpterm | N | N | Y |
| tptypes | Y | Y | Y |
| tpunadvertise | Y | Y | N |
| tpunsubscribe | Y | Y | Y |
| tpxmltofml | Y | Y | Y |
| tpxmltofml32 | Y | Y | Y |

The remaining state tables are for both clients and servers, unless otherwise noted. Keep in mind that because some functions cannot be called by both clients and servers (for example, `tpinit()`), certain state transitions shown below may not be possible for both process types. The above table should be consulted to determine whether the process in question is allowed to call a particular function.

The following state table indicates whether or not a thread in a client process has been initialized and registered with the transaction manager. Note that this table assumes the use of `tpinit()`, which is optional in single-context mode. That is, a single-context client may implicitly join an application by issuing one of many ATMI functions (for example, `tpconnect()` or `tpcall()`). A client must use `tpinit()` when one of the following is true:

- Application authentication is required. (See `tpinit(3c)` and the description of the `SECURITY` keyword in `UBBCONFIG(5)` for details.)
- The client wants to access an XA-compliant resource manager directly. (See `tpinit(3c)` for details.)
- The client wants to create multiple application associations.

A server is placed in the initialized state by the BEA Tuxedo ATMI system's `main()` before its `tpsvrinit()` function is invoked, and it is placed in the uninitialized state by the BEA Tuxedo ATMI system's `main()` after its `tpsvrdone()` function has returned. Note that in all of the state tables shown below, an error return from a function causes the thread to remain in the same state, unless otherwise noted.

**Table 5 Thread Initialization States**

| Function | States | |
|---|---|---|
| | Uninitialize $I_0$ | Initialize $I_1$ |
| `tpalloc` | $I_0$ | $I_1$ |
| `tpchkauth` | $I_0$ | $I_1$ |
| `tpfree` | $I_0$ | $I_1$ |
| `tpgetctxt` | $I_0$ | $I_1$ |
| `tpinit` | $I_1$ | $I_1$ |
| `tprealloc` | $I_0$ | $I_1$ |
| `tpsetctxt` (set to a non-NULL context) | $I_1$ | $I_1$ |
| `tpsetctxt` (with the `TPNULLCONTEXT` context set) | $I_0$ | $I_0$ |
| `tpsetunsol` | $I_0$ | $I_1$ |
| `tpterm` (in this thread) | $I_0$ | $I_0$ |

**Table 5 Thread Initialization States (Continued)**

| Function | States | |
|---|---|---|
| | Uninitialize $I_0$ | Initialize $I_1$ |
| `tpterm` (in a different thread of this context) | $I_0$ | $I_0$ |
| `tptypes` | $I_0$ | $I_1$ |
| All other ATMI functions | $I_1$ | $I_1$ |

The remaining state tables assume a precondition of state $I_1$ (regardless of whether a process arrived in this state via `tpinit()`, `tpsetctxt()`, or the BEA Tuxedo ATMI system's `main()`).

Table 6 indicates the state of a client or server with respect to whether or not a resource manager associated with the process has been initialized.

**Table 6 Resource Management States**

| Function | States | |
|---|---|---|
| | Closed $R_0$ | Open $R_1$ |
| `tpopen` | $R_1$ | $R_1$ |
| `tpclose` | $R_0$ | $R_0$ |
| `tpbegin` | | $R_1$ |
| `tpcommit` | | $R_1$ |
| `tpabort` | | $R_1$ |
| `tpsuspend` | | $R_1$ |
| `tpresume` | | $R_1$ |

**Table 6 Resource Management States (Continued)**

| Function | States | |
|---|---|---|
| | Closed $R_0$ | Open $R_1$ |
| `tpservice` with flag `TPTRAN` | | $R_1$ |
| All other ATMI functions | $R_0$ | $R_1$ |

Table 7 indicates the state of a process with respect to whether or not the process is associated with a transaction. For servers, transitions to states $T_1$and $T_2$ assume a precondition of state $R_1$ (for example, `tpopen()` has been called with no subsequent call to `tpclose()` or `tpterm()`).

**Table 7 Transaction State of Application Association**

| Function | State | | |
|---|---|---|---|
| | Not in Transaction $T_0$ | Initiator $T_1$ | Participant $T_2$ |
| `tpbegin` | | | |
| `tpabort` | | $T_0$ | |
| `tpcommit` | | $T_0$ | |
| `tpsuspend` | | $T_0$ | |
| `tpresume` | $T_1$ | $T_0$ | |
| `tpservice` with flag `TPTRAN` | $T_2$ | | |
| `tpservice` (not in transaction mode) | $T_0$ | | |
| `tpreturn` | $T_0$ | | $T_0$ |
| `tpforward` | $T_0$ | | $T_0$ |
| `tpclose` | $R_0$ | | |

**Table 7 Transaction State of Application Association (Continued)**

| Function | State | | |
|---|---|---|---|
| | Not in Transaction $T_0$ | Initiator $T_1$ | Participant $T_2$ |
| `tpterm` | $I_0$ | $T_0$ | |
| All other ATMI functions | $T_0$ | $T_1$ | $T_2$ |

Table 8 indicates the state of a single request descriptor returned by `tpacall()`.

**Table 8 Asynchronous Request Descriptor States**

| Function | States | |
|---|---|---|
| | No Descriptor $A_0$ | Valid Descriptor $A_1$ |
| `tpacall` | $A_1$ | |
| `tpgetrply` | | $A_0$ |
| `tpcancel` | | $A_0$ [a] |
| `tpabort` | $A_0$ | $A_0$ [b] |
| `tpcommit` | $A_0$ | $A_0$ [b] |
| `tpsuspend` | $A_0$ | $A_1$ [c] |
| `tpreturn` | $A_0$ | $A_0$ |
| `tpforward` | $A_0$ | $A_0$ |
| `tpterm` | $I_0$ | $I_0$ |
| All other ATMI functions | $A_0$ | $A_1$ |

**Note:** [a] This state change occurs only if the descriptor is not associated with the caller's transaction.

[b] This state change occurs only if the descriptor is associated with the caller's transaction.

[c] If the descriptor is associated with the caller's transaction, then `tpsuspend()` returns a protocol error.

Table 9 indicates the state of a connection descriptor returned by `tpconnect()` or provided by a service invocation in the `TPSVCINFO` structure. For primitives that do not take a connection descriptor, the state changes apply to all connection descriptors, unless otherwise noted.

The states are as follows:

- $C_0$—No descriptor
- $C_1$—`tpconnect()` descriptor send-only
- $C_2$—`tpconnect()` descriptor receive-only
- $C_3$—`TPSVCINFO` descriptor send-only
- $C_4$—`TPSVCINFO` descriptor receive-only

**Table 9 Connection Request Descriptor States**

| Function/Event | States | | | | |
|---|---|---|---|---|---|
| | $C_0$ | $C_1$ | $C_2$ | $C_3$ | $C_4$ |
| `tpconnect` with `TPSENDONLY` | $C_1$ [a] | | | | |
| `tpconnect` with `TPRECVONLY` | $C_2$ [a] | | | | |
| `tpservice` with flag `TPSENDONLY` | $C_3$ [b] | | | | |
| `tpservice` with flag `TPRECVONLY` | $C_4$ [b] | | | | |
| `tprecv`/no event | | | $C_2$ | | $C_4$ |
| `tprecv/TPEV_SENDONLY` | | | $C_1$ | | $C_3$ |
| `tprecv/TPEV_DISCONIMM` | | | $C_0$ | | $C_0$ |
| `tprecv/TPEV_SVCERR` | | | $C_0$ | | |
| `tprecv/TPEV_SVCFAIL` | | | $C_0$ | | |

**Table 9 Connection Request Descriptor States (Continued)**

| Function/Event | States | | | | |
|---|---|---|---|---|---|
| | $C_0$ | $C_1$ | $C_2$ | $C_3$ | $C_4$ |
| tprecv/TPEV_SVCSUCC | | | $C_0$ | | |
| tpsend/no event | | $C_1$ | | $C_3$ | |
| tpsend with flag TPRECVONLY | | $C_2$ | | $C_4$ | |
| tpsend/TPEV_DISCONIMM | | $C_0$ | | $C_0$ | |
| tpsend/TPEV_SVCERR | | $C_0$ | | | |
| tpsend/TPEV_SVCFAIL | | $C_0$ | | | |
| tpterm (client only) | $C_0$ | $C_0$ | | | |
| tpcommit (originator only) | $C_0$ | $C_0$ [c] | $C_0$ [c] | | |
| tpsuspend (originator only) | $C_0$ | $C_1$ [d] | $C_2$ [d] | | |
| tpabort (originator only) | $C_0$ | $C_0$ [c] | $C_0$ [c] | | |
| tpdiscon | | $C_0$ | $C_0$ | | |
| tpreturn (CONV server) | | $C_0$ | $C_0$ | $C_0$ | $C_0$ |
| tpforward (CONV server) | | $C_0$ | $C_0$ | $C_0$ | $C_0$ |
| All other ATMI functions | $C_0$ | $C_1$ | $C_2$ | $C_3$ | $C_4$ |

**Note:** [a] If process is in transaction mode and TPNOTRAN is not specified, the connection is in transaction mode.

[b] If the TPTRAN flag is set, the connection is in transaction mode.

[c] If the connection is not in transaction mode, no state change.

[d] If the connection is in transaction mode, then tpsuspend() returns a protocol error.

## TX State Transitions

The BEA Tuxedo ATMI system ensures that a process calls the TX functions in a legal sequence. When an illegal state transition is attempted (that is, a call from a state with a blank transition entry), the called function returns TX_PROTOCOL_ERROR. The legal states and transitions for the TX functions are shown in Table 10. Calls that return failure do not make state transitions, unless they are described by specific state table entries. Any BEA Tuxedo ATMI system client or server is allowed to use the TX functions.

The states are defined below:

- $S_0$: No RMs have been opened or initialized. An application association cannot start a global transaction until it has successfully called tx_open.
- $S_1$: An application association has opened its RM but is not in a transaction. Its transaction_control characteristic is TX_UNCHAINED.
- $S_2$: An application association has opened its RM but is not in a transaction. Its transaction_control characteristic is TX_CHAINED.
- $S_3$: An application association has opened its RM and is in a transaction. Its transaction_control characteristic is TX_UNCHAINED.
- $S_4$: An application association has opened its RM and is in a transaction. Its transaction_control characteristic is TX_CHAINED.

**Table 10 TX Function States and Transitions**

| Function | States | | | | |
|---|---|---|---|---|---|
| | $S_0$ | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
| tx_begin | | $S_3$ | $S_4$ | | |
| tx_close | $S_0$ | $S_0$ | $S_0$ | | |
| tx_commit -> TX_SET1 | | | | $S_1$ | $S_4$ |
| tx_commit -> TX_SET2 | | | | | $S_2$ |
| tx_info | | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
| tx_open | $S_1$ | $S_1$ | $S_2$ | $S_3$ | $S_4$ |

**Table 10 TX Function States and Transitions (Continued)**

| Function | States | | | | |
|---|---|---|---|---|---|
| | $S_0$ | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
| tx_rollback → TX_SET1 | | | | $S_1$ | $S_4$ |
| tx_rollback → TX_SET2 | | | | | $S_2$ |
| tx_set_commit_return | | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
| tx_set_transaction_control control = TX_CHAINED | | $S_2$ | $S_2$ | $S_4$ | $S_4$ |
| tx_set_transaction_control control = TX_UNCHAINED | | $S_1$ | $S_1$ | $S_3$ | $S_3$ |
| tx_set_transaction_timeout | | $S_1$ | $S_2$ | $S_3$ | $S_4$ |

- TX_SET1 denotes any of the following: TX_OK, TX_ROLLBACK, TX_MIXED, TX_HAZARD, or TX_COMMITTED. TX_ROLLBACK is not returned by tx_rollback() and TX_COMMITTED is not returned by tx_commit().
- TX_SET2 denotes any of the following: TX_NO_BEGIN, TX_ROLLBACK_NO_BEGIN, TX_MIXED_NO_BEGIN, TX_HAZARD_NO_BEGIN, or TX_COMMITTED_NO_BEGIN. TX_ROLLBACK_NO_BEGIN is not returned by tx_rollback() and TX_COMMITTED_NO_BEGIN is not returned by tx_commit().
- If TX_FAIL is returned on any call, the application process is in an undefined state with respect to the above table.
- When tx_info() returns either TX_ROLLBACK_ONLY or TX_TIMEOUT_ROLLBACK_ONLY in the transaction state information, the transaction is marked rollback-only and will be rolled back whether the application program calls tx_commit() or tx_rollback().

## See Also

buffer(3c), tpadvertise(3c), tpalloc(3c), tpbegin(3c), tpcall(3c), tpconnect(3c), tpgetctxt(3c), tpinit(3c), tpopen(3c), tpservice(3c), tpsetctxt(3c), tuxtypes(5), typesw(5)

## AEMsetblockinghook(3c)

### Name

`AEMsetblockinghook()`—Establishes an application-specific blocking hook function.

### Synopsis

```
#include <atmi.h>
int AEMsetblockinghook(_TM_FARPROC)
```

### Description

`AEMsetblockinghook()` is an "ATMI Extension for Mac" that allows a Mac task to install a new function which the ATMI networking software uses to implement blocking ATMI calls. It takes a pointer to the procedure instance address of the blocking function to be installed.

A default function, by which blocking ATMI calls are handled, is included. The function `AEMsetblockinghook()` gives the application the ability to execute its own function at "blocking" time in place of the default function. If called with a NULL pointer, the blocking hook function is reset to the default function.

When an application invokes a blocking ATMI operation, the operation is initiated and then a loop is entered which is equivalent to the following pseudocode:

```
for(;;) {
        execute operation in non-blocking mode
        if error
               break;
        if operation complete
               break;
        while(BlockingHook())
                 ;
}
```

### Return Values

`AEMsetblockinghook()` returns a pointer to the procedure-instance of the previously installed blocking function. The application or library that calls the `AEMsetblockinghook()` function should save this return value so that it can be restored if necessary. (If "nesting" is not important, the application may simply discard the value returned by `AEMsetblockinghook()` and eventually use `AEMsetblockinghook`(NULL) to restore the default mechanism.)

AEMsetblockinghook() returns NULL on error and sets tperrno to indicate the error condition.

### Errors

Under failure, AEMsetblockinghook() sets tperrno to the following value:

**[**TPEPROTO**]**
AEMsetblockinghook() was called while a blocking operation was in progress.

### Portability

This interface is supported only in Mac clients.

### Notices

The blocking function is reset after tpterm(3c) is called by the application.

## AEOaddtypesw(3c)

### Name

AEOaddtypesw()—Installs or replaces a user-defined buffer type at execution time.

### Synopsis

```
#include <atmi.h>
#include <tmtypes.h>

int FAR PASCAL AEOaddtypesw(TMTYPESW *newtype)
```

### Description

AEOaddtypesw() is an "ATMI Extension for OS/2" that allows an OS/2 client to install a new, or replace an existing, user-defined buffer type at execution time. The argument to this function is a pointer to a TMTYPESW structure that contains the information for the buffer type to be installed.

If the type() and the subtype() match an existing buffer type already installed, then all the information is replaced with the new buffer type. If the information does not match the type() and the subtype() fields, then the new buffer type is added to the existing types registered with the BEA Tuxedo ATMI system. For new buffer types, make sure that the WSH and other BEA Tuxedo ATMI system processes involved in the call processing have been built with the new buffer type.

The function pointers in the TMTYPESW array should appear in the Module Definition file of the application in the EXPORTS section.

The application can also use the BEA Tuxedo ATMI system's defined buffer type routines. The application and the BEA Tuxedo ATMI system's buffer routines can be intermixed in one user defined buffer type.

## Return Values

Upon success, AEOaddtypesw() returns the number of user buffer types in the system Upon failure, AEOaddtypesw() returns -1 and sets tperrno to indicate the error condition.

## Errors

Upon failure, AEOaddtypesw() sets tperrno to one of the following values:

**[**TPEINVAL**]**
AEOaddtypesw() was called and the type parameter was NULL.

**[**TPESYSTEM**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

## Portability

This interface is supported only in Windows clients. The preferred way to install a type switch is to add it to the BEA Tuxedo ATMI system type switch DLL. Please refer to *Setting Up a BEA Tuxedo Application* for more information.

## Notices

FAR PASCAL is used only for the 16-bit OS/2 environment.

## Examples

```
#include <os2.h>
#include <atmi.h>
#include <tmtypes.h>

int FAR PASCAL Nfinit(char FAR *, long);
int (FAR PASCAL * lpFinit)(char FAR *, long);
int FAR PASCAL Nfreinit(char FAR *, long);
int (FAR PASCAL * lpFreinit)(char FAR *, long);
int FAR PASCAL Nfuninit(char FAR *, long);
int (FAR PASCAL * lpFuninit)(char FAR *, long);

TMTYPESW        newtype =
```

```
{
“MYFML”,          “”,             1024,           NULL,           NULL,
NULL,           _fpresend,      _fpostsend,     _fpostrecv,     _fencdec,
_froute
};

   newtype.initbuf = Nfinit;
   newtype.reinitbuf = Nfreinit;
   newtype.uninitbuf = Nfuninit;

   if(AEOaddtypesw(newtype) == -1) {
          userlog(“AEOaddtypesw failed %s”, tpstrerror(tperrno));
   }
int
FAR PASCAL
Nfinit(char FAR *ptr, long len)
{
      ......
      return(1);
}

int
FAR PASCAL
Nfreinit(char FAR *ptr, long len)
{
      ......
      return(1);
}

int
FAR PASCAL
Nfuninit(char FAR *ptr, long mdlen)
{
       ......
       return(1);
}
```

The application Module Definition File:

```
; EXAMPLE.DEF file

NAME         EXAMPLE

DESCRIPTION  'EXAMPLE for OS/2'

EXETYPE      OS/2


EXPORTS
```

```
Nfinit
Nfreinit
Nfuninit
 ....
```

### See Also

`buildwsh(1)`, `buffer(3c)`, `typesw(5)`

## AEPisblocked(3c)

### Name

`AEPisblocked()`—Determines if a blocking call is in progress.

### Synopsis

```
#include <atmi.h>
int far pascal AEPisblocked(void)
```

### Description

`AEPisblocked()` is an "ATMI Extension for OS/2 Presentation Manager" that allows a OS/2 PM task to determine if it is executing while waiting for a previous blocking call to complete.

### Return Values

If there is an outstanding blocking function awaiting completion, `AEPisblocked()` returns 1. Otherwise, it returns 0.

### Errors

No errors are returned.

### Portability

This interface is supported only in OS/2 PM clients.

### Comments

Although a blocking ATMI call appears to an application as though it "blocks," the OS/2 PM ATMI DLL has to relinquish the processor to allow other applications to run. This means that it is possible for the application which issued the blocking call to be reentered, depending on the message(s) it receives. In this instance, the `AEPisblocked()` function can be used to ascertain whether the task has been reentered while waiting for an outstanding blocking call to complete. Note that ATMI prohibits more than one outstanding call per thread.

### See Also

AEMsetblockinghook(3c)

## AEWsetunsol(3c)

### Name

AEWsetunsol()—Posts a Windows message for BEA Tuxedo ATMI unsolicited event.

### Synopsis

```
#include <windows.h>
#include <atmi.h>
int far pascal AEWsetunsol(HWND hWnd, WORD wMsg)
```

### Description

In certain Microsoft Windows programming environments, it is natural and convenient for the BEA Tuxedo ATMI system's unsolicited messages to be posted to the Windows event message queue.

AEWsetunsol() controls which window to notify, *hWnd*, and which Windows message type to post, *wMsg*. When a BEA Tuxedo ATMI unsolicited message arrives, a Windows message is posted. lParam() is set to the BEA Tuxedo ATMI system buffer pointer, or zero if none. If lParam() is non-zero, the application must call tpfree() to release the buffer.

If *wMsg* is zero, any future unsolicited messages will be logged and ignored.

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to AEWsetunsol().

### Return Values

Upon failure, AEWsetunsol() returns -1 and sets tperrno to indicate the error condition.

### Errors

Upon failure, AEWsetunsol() sets tperrno to one of the following values:

[TPESYSTEM]
: A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

[TPEOS]
: An operating system error has occurred.

### Portability

This interface is supported only in Microsoft Windows clients.

### Notices

`AEWsetunsol()` posting of Windows messages may not be activated simultaneously with a `tpsetunsol()` callback routine. The most recent `tpsetunsol()` or `AEWsetunsol()` request controls how unsolicited messages will be handled.

### See Also

`tpsetunsol(3c)`

## buffer(3c)

### Name

`buffer()`—Semantics of elements in `tmtype_sw_t`.

### Synopsis

```
int     /* Initialize a new data buffer */
_tminitbuf(char *ptr, long len)
int    /* Reinitialize a reallocated data buffer */
_tmreinitbuf(char *ptr, long len)
int    /* Uninitialize a data buffer to be freed */
_tmuninitbuf(char *ptr, long len)
long   /* Process buffer before sending */
_tmpresend(char *ptr, long dlen, long mdlen)
void   /* Process buffer after sending */
_tmpostsend(char *ptr, long dlen, long mdlen)
long   /* Process buffer after receiving */
_tmpostrecv(char *ptr, long dlen, long mdlen)
long   /* Encode/decode a buffer to/from a transmission format */
_tmencdec(int op, char *encobj, long elen, char *obj, long olen)
int    /* Determine server group for routing based on data */
_tmroute(char *routing_name, char *service, char *data, long \  len, char *group)
int    /* Evaluate boolean expression on buffer's data */
_tmfilter(char *ptr, long dlen, char *expr, long exprlen)
int    /* Extract buffer's data based on format string */
_tmformat(char *ptr, long dlen, char *fmt, char *result, long \  maxresult)
long   /* Process buffer before sending, possibly generating copy */
_tmpresend2(char *iptr, long ilen, long mdlen, char *optr, long olen, long *flags
)
long  /* Multibyte code-set encoding conversion */
```

```
_tmconvmb(char *ibufp, long ilen, char *enc_name, char *obufp, long olen, long
    *flags)
```

## Description

This page describes the semantics of the elements and routines defined in the `tmtype_sw_t` structure. These descriptions are necessary for adding new buffer types to a process buffer type switch, `tm_typesw`. The switch elements are defined in `typesw(5)`. The function names used in this entry are templates for the actual function names defined by the BEA Tuxedo ATMI system as well as by applications adding their own buffer types. The names map to the switch elements very simply: the template names are made by taking each function pointer's element name and prepending `_tm` (for example, the element `initbuf` has the function name `_tminitbuf()`).

The element `type` must be non-NULL and up to 8 characters in length. The element `subtype` can be NULL, a string of up to 16 characters, or the wildcard character, "*". If `type` is not unique in the switch, then `subtype` must be used; the combination of `type` and `subtype` must uniquely identify an element in the switch.

A given type can have multiple subtypes. If all subtypes are to be treated the same for a given type, then the wildcard character, "*", can be used. Note that the function `tptypes()` can be used to determine a buffer's type and subtype if subtypes need to be distinguished. If some subset of the subtypes within a particular type are to be treated individually, and the rest are to be treated identically, then those which are to be singled out with specific subtype values should appear in the switch before the subtype designated with the wildcard. Thus, searching for types and subtypes in the switch is done from top to bottom, and the wildcard subtype entry accepts any "leftover" type matches.

`dfltsize()` is used when allocating or reallocating a buffer. The larger of `dfltsize()` and the routines' *`size`* parameter is used to create or reallocate a buffer. For some types of structures, like a fixed sized C structure, the buffer size should equal the size of the structure. If `dfltsize()` is set to this value, then the caller may not need to specify the buffer's length to routines in which a buffer is passed. `dfltsize()` can be 0 or less; however, if `tpalloc()` or `tprealloc()` is called and its *`size`* parameter is also less than or equal to 0, then the routine will fail. It is not recommended to set `dfltsize()` to a value less than 0.

## Routine Specifics

The names of the functions specified below are template names used within the BEA Tuxedo ATMI system. Any application adding new routines to the buffer type switch must use names that correspond to real functions, either provided by the application or library routines. If a NULL function pointer is stored in a buffer type switch entry, the BEA Tuxedo ATMI system calls a default function that takes the correct number and type of arguments, and returns a default value.

## _tminitbuf

`_tminitbuf()` is called from within `tpalloc()` after a buffer has been allocated. It is passed a pointer to the new buffer, *`ptr`*, along with its size so that the buffer can be initialized appropriately. *`len`* is the larger of the length passed into `tpalloc()` and the default specified in `dfltsize()` in that type's switch entry. Note that *`ptr`* will never be NULL due to the semantics of `tpalloc()` and `tprealloc()`. Upon successful return, *`ptr`* is returned to the caller of `tpalloc()`.

If a single switch entry is used to manipulate many subtypes, then the writer of `_tminitbuf()` can use `tptypes()` to determine the subtype.

If no buffer initialization needs to be performed, specify a NULL function pointer.

Upon success, `_tminitbuf()` returns 1. If the function fails, it returns -1 causing `tpalloc()` to also return failure setting `tperrno` to `TPESYSTEM`.

## _tmreinitbuf

`_tmreinitbuf()` behaves the same as `_tminitbuf()` except it is used to reinitialize a reallocated buffer. It is called from within `tprealloc()` after the buffer has been reallocated.

If no buffer reinitialization needs to be performed, specify a NULL function pointer.

Upon success, `_tmreinitbuf()` returns 1. If the function fails, it returns -1 causing `tprealloc()` to also return failure setting `tperrno` to `TPESYSTEM`.

## _tmuninitbuf

`_tmuninitbuf()` is called by `tpfree()` before the data buffer is freed. `_tmuninitbuf()` is passed a pointer to the application portion of a data buffer, along with its size, and can be used to clean up any structures or state information associated with that buffer. *`ptr`* will never be NULL due to `tpfree()`'s semantics. Note that `_tmuninitbuf()` should not free the buffer itself. The `tpfree()` function is called automatically for any `FLD_PTR` fields in the data buffer.

If no processing needs to be performed before freeing a buffer, specify a NULL function pointer.

Upon success, `_tmuninitbuf()` returns 1. If the function fails, it returns -1 causing `tpfree()` to print a log message.

## _tmpresend

`_tmpresend()` is called before a buffer is sent in `tpcall()`, `tpacall()`, `tpconnect()`, `tpsend()`, `tpbroadcast()`, `tpnotify()`, `tpreturn()`, or `tpforward()`. It is also called after `_tmroute()` but before `_tmencdec()`. If *`ptr()`* is non-NULL, preprocessing is performed on a buffer before it is sent. `_tmpresend()`'s first argument, *`ptr`*, is the application data buffer passed

into the send call. Its second argument, *dlen*, is the data's length as passed into the send call. Its third argument, *mdlen*, is the actual size of the buffer in which the data resides.

One important requirement on this function is that it ensures that when the function returns, the data pointed to by *ptr* can be sent "as is." That is, since _tmencdec() is called only if the buffer is being sent to a dissimilar machine, _tmpresend() must ensure upon return that no element in *ptr'*s buffer is a pointer to data that is not contiguous to the buffer.

If no preprocessing needs to be performed on the data and the amount of data the caller specified is the same as the amount that should be sent, specify a NULL function pointer. The default routine returns *dlen* and does nothing to the buffer.

If _tmpresend2() is not NULL, _tmpresend() is not called and _tmpresend2() is called in its place.

Upon success, _tmpresend() returns the amount of data to be sent. If the function fails, it returns -1 causing _tmpresend()'s caller to also return failure setting tperrno to TPESYSTEM.

## _tmpostsend

_tmpostsend() is called after a buffer is sent in tpcall(), tpbroadcast(), tpnotify(), tpacall(), tpconnect(), or tpsend(). This routine allows any post-processing to be performed on a buffer after it is sent and before the function returns. Because the buffer passed into the send call should not be different upon return, _tmpostsend() is called to repair a buffer changed by _tmpresend(). This function's first argument, *ptr*, points to the data sent as a result of _tmpresend(). The data's length, as returned from _tmpresend(), is passed in as this function's second argument, *dlen*. The third argument, *mdlen*, is the actual size of the buffer in which the data resides. This routine is called only when *ptr* is non-NULL.

If no post-processing needs to be performed, specify a NULL function pointer.

## _tmpostrecv

_tmpostrecv() is called after a buffer is received, and possibly decoded, in tpgetrply(), tpcall(), tprecv(), or in the BEA Tuxedo ATMI system's server abstraction, and before it is returned to the application. If *ptr* is non-NULL, _tmpostrecv() allows post-processing to be performed on a buffer after it is received and before it is given to the application. Its first argument, *ptr*, points to the data portion of the buffer received. Its second argument, *dlen*, specifies the data's size coming in to _tmpostrecv(). The third argument, *mdlen*, specifies the actual size of the buffer in which the data resides.

If _tmpostrecv() changes the data length in post-processing, it must return the data's new length. The length returned is passed up to the application in a manner dependent on the call used

(for example, tpcall() sets the data length in one of its arguments for the caller to check upon return).

The buffer's size might not be large enough for post-processing to succeed. If more space is required, _tmpostrecv() returns the negative absolute value of the desired buffer size. The calling routine then resizes the buffer, and calls _tmpostrecv() a second time.

If no post-processing needs to be performed on the data and the amount of data received is the same as the amount that should be returned to the application, specify a NULL function pointer. The default routine returns *dlen* and does nothing to the buffer.

On success, _tmpostrecv() returns the size of the data the application should be made aware of when the buffer is passed up from the corresponding receive call. If the function fails, it returns -1 causing _tmpostrecv()'s caller to return failure, setting tperrno to TPESYSTEM.

## _tmencdec

_tmencdec() is used to encode/decode a buffer sent/received over a network to/from a machine having different data representations. The BEA Tuxedo ATMI system recommends the use of XDR; however, any encoding/decoding scheme can be used that obeys the semantics of this routine.

This function is called by tpcall(), tpacall(), tpbroadcast(), tpnotify(), tpconnect(), tpsend(), tpreturn(), or tpforward() to encode the caller's buffer only when it is being sent to an "unlike" machine. In these calls, _tmencdec() is called after both _tmroute() and _tmpresend(), respectively. Recall from the description of _tmpresend() that the buffer passed into _tmencdec() contains no pointers to data that is not contiguous to the buffer.

On the receiving end, tprecv(), tpgetrply(), the receive half of tpcall() and the server abstraction all call _tmencdec() to decode a buffer after they have received it from an "unlike" machine but before calling _tmpostrecv().

_tmencdec()'s first argument, *op*, specifies whether the function is encoding or decoding data. *op* can be one of TMENCODE or TMDECODE.

When *op* is TMENCODE, *encobj* points to a buffer allocated by the BEA Tuxedo ATMI system where the encoded version of the data will be copied. The unencoded data resides in *obj*. That is, when *op* is TMENCODE, _tmencdec() transforms *obj* to its encoded format and places the result in *encobj*. The size of the buffer pointed to by *encobj* is specified by *elen* and is at least four times the size of the buffer pointed to by *obj* whose length is *olen*. *olen* is the length returned by _tmpresend. _tmencdec() returns the size of the encoded data in *encobj* (that is, the

amount of data to actually send). _tmencdec() should not free either of the buffers passed into the function.

When *op* is TMDECODE, *encobj* points to a buffer allocated by the BEA Tuxedo ATMI system where the encoded version of the data resides as read off a communication endpoint. The length of the buffer is *elen*. *obj* points to a buffer that is at least the same size as the buffer pointed to by *encobj* into which the decoded data is copied. The length of *obj* is *olen*. As *obj* is the buffer ultimately returned to the application, this buffer may be grown by the BEA Tuxedo ATMI system before calling _tmencdec() to ensure that it is large enough to hold the decoded data. _tmencdec() returns the size of the decoded data in *obj*. After _tmencdec() returns, _tmpostrecv() is called with *obj* passed as its first argument, _tmencdec()'s return value as its second, and *olen* as its third. _tmencdec() should not free either of the buffers passed into the function.

_tmencdec() is called only when non-NULL data needs to be encoded or decoded.

If no encoding or decoding needs to be performed on the data even when dissimilar machines exist in the network, specify a NULL function pointer. The default routine returns either *olen* (*op* equals TMENCODE) or *elen* (*op* equals TMDECODE).

On success, _tmencdec() returns a non-negative length as described above. If the function fails, it returns -1 causing _tmencdec()'s caller to return failure, setting tperrno to TPESYSTEM.

## _tmroute

The default for message routing is to route a message to any available server group that offers the desired service. Each service entry in the UBBCONFIG file can specify the logical name of some routing criteria for the service using the ROUTING parameter. Multiple services can share the same routing criteria. In the case that a service has a routing criteria name specified, _tmroute() is used to determine the server group to which a message is sent based on data in the message. This mapping of data to server group is called "data-dependent routing." _tmroute() is called before a buffer is sent (and before _tmpresend() and _tmencdec() are called) in tpcall(), tpacall(), tpconnect(), and tpforward().

*routing_name* is the logical name of the routing criteria (as specified in the UBBCONFIG file) and is associated with every service that needs data dependent routing. *service* is the name of the service for which the request is being made. The parameter *data* points to the data that is being transmitted in the request and *len* is its length. Unlike the other routines described in these pages, _tmroute() is called even when *ptr* is NULL. The *group* parameter is used to return the name of the group to which the request should be routed. This group name must match one of the group names listed in the UBBCONFIG file (and one that is active at the time the group is chosen). If the

request can go to any available server providing the specified service, *group* should be set to the NULL string and the function should return 1.

If data dependent routing is not needed for the buffer type, specify a NULL function pointer. The default routine sets *group* to the NULL string and returns 1.

Upon success, _tmroute() returns 1. If the function fails, it returns -1 causing _tmroute()'s caller to also return failure; as a result, tperrno is set to TPESYSTEM. If _tmroute() fails because a requested server or service is not available, tperrno is set to TPENOENT.

If *group* is set to the name of an invalid server group, the function calling _tmroute() will return an error and set tperrno to TPESYSTEM.

## _tmfilter

_tmfilter() is called by the EventBroker server to analyze the contents of a buffer posted by tppost(). An expression provided by the subscriber (tpsubscribe()) is evaluated with respect to the buffer's contents. If the expression is true, _tmfilter() returns 1 and the EventBroker performs the subscription's notification action. Otherwise, if _tmfilter() returns 0, the EventBroker does not consider this posting a "match" for the subscription.

If *exprlen* is -1, *expr* is interpreted as a NULL-terminated character string. Otherwise *expr* is interpreted as *exprlen* bytes of binary data. An *exprlen* of 0 indicates no expression.

If filtering does not apply to this buffer type, specify a NULL function pointer. The default routine returns 1 if there is no expression or if *expr* is an empty NULL-terminated string. Otherwise the default routine returns 0.

## _tmformat

_tmformat() is called by the EventBroker server to convert a buffer's data into a printable string, based on a format specification named *fmt*. The EventBroker converts posted buffers to strings as input for *userlog* or *system* notification actions.

The output is stored as a character string in the memory location pointed to by *result*. Up to *maxresult* bytes are written in *result*, including a terminating NULL character. If *result* is not large enough, _tmformat() truncates its output. The output string is always NULL terminated.

On success, _tmformat() returns a non-negative integer. 1 means success, 2 means the output string is truncated. If the function fails, it returns -1 and stores an empty string in *result*.

If formatting does not apply to this buffer type, specify a NULL function pointer. The default routine succeeds and returns an empty string in *result*.

## _tmpresend2

_tmpresend2() is called before a buffer is sent in tpcall(), tpacall(), tpconnect(), tpsend(), tpbroadcast(), tpnotify(), tpreturn(), and tpforward(). It is also called after _tmroute() but before _tmencdec(). If *iptr* is not NULL, preprocessing is performed on a buffer before the buffer is sent.

The first argument to _tmpresend2(), *iptr*, is the application data buffer passed into the send call. The second argument, *ilen*, is the length of the data as passed into the send call. The third argument, *mdlen*, is the actual size of the buffer in which the data resides.

Unlike _tmpresend(), _tmpresend2() receives a pointer, *optr*, which is used to pass a pointer to a buffer into which the data in *iptr* can be placed, after any required processing is done. Use this pointer if you want to use a new buffer for the data modified by _tmpresend2() instead of modifying the input buffer. The fifth argument, *olen*, is the size of the *optr* buffer. The sixth argument, *flags*, tells _tmpresend2() whether the buffer being processed is the parent buffer (the one being sent). The *flags* argument is returned by _tmpresend2() to indicate the results of processing.

The size of the *optr* buffer may not be large enough for successful postprocessing. If more space is required, _tmpresend2() returns the negative absolute value of the desired buffer size. All *olen* bytes of the *optr* buffer are preserved. The calling routine then resizes the buffer and calls _tmpresend2() a second time.

If no postprocessing needs to be performed on the data, and the amount of data received is the same as the amount that should be returned to the application, specify a NULL function pointer. The default routine returns *ilen* and does not modify the buffer.

The following is a valid flag on input to _tmpresend2():

**[**TMPARENT**]**
: This is the parent buffer (the one being sent).

The flags returned in *flags* specify the results of _tmpresend2(). Possible values are:

**[**TMUSEIPTR**]**
: _tmpresend2() was successful: the processed data is in the buffer referenced by *iptr*, and the return value contains the length of the data to be sent.

**[**TMUSEOPTR**]**
: _tmpresend2() was successful: the processed data is in the buffer referenced by *optr*, and the return value contains the length of the data to be sent.

If TMUSEOPTR is returned, the processing done after messages are transmitted is different from the processing done by _tmpresend(): the *iptr* buffer remains unchanged and _tmpostsend() is not called. If TMUSEIPTR is returned, _tmpostsend() is called, as it is called for _tmpresend(). It is the responsibility of the caller to allocate and to free or cache the *optr* buffer.

There are several reasons why you may want to use this approach for a typed buffer:

- The buffer created by processing for transmission is larger than the maximum length allowed for the input buffer.
- Undoing the processing to prepare a buffer for transmission is so complicated that it is easier to copy the data to a different buffer.

The _tmpresend2() function ensures that when a function returns, the data in the buffer to be sent can be sent without further processing. Because _tmencdec() is called only if the buffer is being sent to a dissimilar machine, _tmpresend2() ensures, upon return, that all data is stored contiguously in the buffer to be sent.

If no preprocessing needs to be performed on the data, and the amount of data specified by the caller is the same as the amount that should be sent, specify a NULL function pointer for _tmpresend2() in the buffer type switch. If _tmpresend2() is NULL, _tmpresend() is called by default.

Upon success, _tmpresend2() returns the amount of data to be sent or, if a larger buffer is needed, the negative absolute value of the desired buffer size. If the function fails, it returns -1, causing the caller of _tmpresend2() to also return failure, setting tperrno to TPESYSTEM.

## _tmconvmb

_tmconvmb() is called after tmpostrecv() to convert multibyte data from a source encoding to a target encoding. The first argument to _tmconvmb(), *ibufp*, is a pointer to a stream of bytes—the multibyte data—to be converted. The second argument, *ilen*, is the number of bytes in *ibufp*. The third argument, *enc_name*, is one of the encoding names used in the processing. For an MBSTRING buffer, the third argument is the target encoding name; for an FML32 buffer, the third argument is the source encoding name.

_tmconvmb() receives a pointer, *obufp*, which is used to pass a pointer to a buffer into which the data in *ibufp* can be placed, after any required code-set encoding conversion is done. Use this pointer if you want to use a new buffer for the data converted by _tmconvmb() instead of modifying the input pointer. The fifth argument, *olen*, is the size of the *obufp* buffer. The *flags* argument is returned by _tmconvmb() to indicate the results of processing.

The size of the *obufp* buffer may not be large enough for successful post processing. If more space is required, _tmconvmb() returns the negative absolute value of the desired buffer size. All *ilen* bytes of the *ibufp* buffer are preserved. The calling routine then resizes the buffer and calls _tmconvmb() a second time.

If no code-set encoding conversion needs to be performed on the data, and the encoding name of the sending process is the same as the encoding name of the receiving process, specify a NULL function pointer. The default routine returns *ilen* and does not convert the buffer. If this function does not know how to convert the code-set encoding, it returns -1.

The value returned in *flags* specifies the result of _tmconvmb(). Possible values are:

**[**TMUSEIPTR**]**

_tmconvmb() was successful: the processed data is in the buffer referenced by *ibufp*, and the return value contains the length of the converted data to be passed to the service.

**[**TMUSEOPTR**]**

_tmconvmb() was successful: the processed data is in the buffer referenced by *obufp*, and the return value contains the length of the data to be converted. It is the responsibility of the caller to allocate and to free or cache the *obufp* buffer.

Upon success, _tmconvmb() returns the amount of data buffer that had code-set encoding conversion or, if a larger buffer is needed, the negative absolute value of the desired buffer size. If the function fails, it returns -1, causing the caller of _tmconvmb() to also return failure, setting tperrno to TPESYSTEM.

### See Also

tpacall(3c), tpalloc(3c), tpcall(3c), tpconnect(3c), tpdiscon(3c), tpfree(3c), tpgetrply(3c), tpgprio(3c), tprealloc(3c), tprecv(3c), tpsend(3c), tpsprio(3c), tptypes(3c), tuxtypes(5)

## catgets(3c)

### Name

catgets()—Reads a program message.

### Synopsis

```
#include <nl_types.h>
char *catgets (nl_catd catd, int set_num, int msg_num, char *s)
```

### Description

catgets() attempts to read message *msg_num*, in set *set_num*, from the message catalogue identified by *catd*. *catd* is a catalogue descriptor returned from an earlier call to catopen(). *s* points to a default message string which will be returned by catgets() if the identified message catalogue is not currently available.

A thread in a multithreaded application may issue a call to catgets() while running in any context state, including TPINVALIDCONTEXT.

### Diagnostics

If the identified message is retrieved successfully, catgets() returns a pointer to an internal buffer area containing the NULL terminated message string. If the call is unsuccessful because the message catalogue identified by *catd* is not currently available, a pointer to *s* is returned.

### See Also

catopen, catclose(3c)

## catopen, catclose(3c)

### Name

catopen(), catclose()—Opens/closes a message catalogue.

### Synopsis

```
#include <nl_types.h>
nl_catd catopen (char *name, int oflag)
int catclose (nl_catd catd)
```

### Description

catopen() opens a message catalogue and returns a catalogue descriptor. *name* specifies the name of the message catalogue to be opened. If *name* contains a "/" then *name* specifies a pathname for the message catalogue. Otherwise, the environment variable NLSPATH is used. If NLSPATH does not exist in the environment, or if a message catalogue cannot be opened in any of the paths specified by NLSPATH, then the default path is used (see nl_types(5)).

The names of message catalogues, and their location in the filestore, can vary from one system to another. Individual applications can choose to name or locate message catalogues according to their own special needs. A mechanism is therefore required to specify where the catalogue resides.

The NLSPATH variable provides both the location of message catalogues, in the form of a search path, and the naming conventions associated with message catalogue files. For example:

```
NLSPATH=/nlslib/%L/%N.cat:/nlslib/%N/%L
```

The metacharacter % introduces a substitution field, where %L substitutes the current setting of the LANG environment variable (see following section), and %N substitutes the value of the *name* parameter passed to catopen(). Thus, in the above example, catopen() will search in /nlslib/$LANG/*name*.cat, then in /nlslib/*name*/$LANG, for the required message catalogue.

NLSPATH will normally be set up on a system wide basis (for example, in /etc/profile) and thus makes the location and naming conventions associated with message catalogues transparent to both programs and users.

The following table lists the full set of metacharacters.

| Metacharacter | Description |
|---|---|
| %N | The value of the name parameter passed to catopen. |
| %L | The value of LANG. |
| %l | The value of the language element of LANG. |
| %t | The value of the territory element of LANG. |
| %c | The value of the codeset element of LANG. |
| %% | A single %. |

The LANG environment variable provides the ability to specify the user's requirements for native languages, local customs and character set, as an ASCII string in the form
LANG=language[_territory[.codeset]]

A user who speaks German as it is spoken in Austria and has a terminal that operates in ISO 8859/1 codeset, would want the setting of the LANG variable to be as follows:

```
LANG=De_A.88591
```

With this setting it should be possible for the user to find relevant catalogues if they exist.

If the LANG variable is not set then the value of LC_MESSAGES as returned by setlocale(3c) is used. If this is NULL then the default path as defined in nl_types(5) is used.

*oflag()* is reserved for future use and should be set to 0. The results of setting this field to any other value are undefined.

`catclose()` closes the message catalogue identified by *catd*.

A thread in a multithreaded application may issue a call to `catopen()` or `catclose()` while running in any context state, including `TPINVALIDCONTEXT`.

### Diagnostics

If successful, `catopen()` returns a message catalogue descriptor for use on subsequent calls to `catgets()` and `catclose()`. Otherwise `catopen()` returns `(nl_catd) -1`. `catclose()` returns 0 if successful, otherwise -1.

### See Also

`catgets(3c)`, `setlocale(3c)`, `nl_types(5)`

## decimal(3c)

### Name

`decimal()`—Decimal conversion and arithmetic routines.

### Synopsis

```
#include "decimal.h"

int
lddecimal(cp, len, np)        /* load a decimal */
char*cp;         /* input: location of compacted format */

int
len;             /* input: length of compacted format */
dec_t*np;        /* output: location of dec_t format */

void
stdecimal(np, cp, len)       /* store a decimal */
dec_t*np;        /* input: location of dec_t format */
char*cp;         /* output: location of compacted format */
int len;         /* input: length of compacted format */

int
deccmp(n1, n2)   /* compare two decimal numbers */
dec_t*n1;        /* input: number to be compared */
dec_t*n2;        /* input: number to be compared */
```

```
int
dectoasc(np, cp, len, right) /* convert dec_t to ascii */
dec_t*np;        /* input: number to be converted */
char*cp;         /* output: number after conversion */
int len;         /* input: length of output string */
int right;       /* input: number of places to right of decimal point */

int
deccvasc(cp, len, np)        /* convert ascii to dec_t */
char*cp;         /* input: number to be converted */
int len;         /* input: maximum length of number to be converted */
dec_t*np;        /* output: number after conversion */

int
dectoint(np, ip)             /* convert int to dec_t */
dec_t*np;        /* input: number to be converted */
int *ip;         /* output: number after conversion */

int
deccvint(in, np) /* convert dec_t to int */
int in;          /* input: number to be converted */
dec_t*np;        /* output: number after conversion */

int
dectolong(np, lngp)         /* convert dec_t to long */
dec_t*np;        /* input: number to be converted */
long*lngp;       /* output: number after conversion */

int
deccvlong(lng, np)        /* convert long to dec_t */
longlng;        /* input: number to be converted */
dec_t*np;       /* output: number after conversion */

int
dectodbl(np, dblp)        /* convert dec_t to double */
dec_t*np;       /* input: number to be converted */
double *dblp;   /* output: number after conversion */

int
deccvdbl(dbl, np)         /* convert double to dec_t */
double *dbl;    /* input: number to be converted */
dec_t*np;       /* output: number after conversion */

int
dectoflt(np, fltp)        /* convert dec_t to float */
dec_t*np;       /* input: number to be converted */
float*fltp;     /* output: number after conversion */

int
deccvflt(flt, np)         /* convert float to dec_t */
```

```
double *flt;    /* input: number to be converted */
dec_t*np;       /* output: number after conversion */

int
decadd(*n1, *n2, *n3)     /* add two decimal numbers */
dec_t*n1;        /* input: addend */
dec_t*n2;        /* input: addend */
dec_t*n3;        /* output: sum */

int
decsub(*n1, *n2, *n3)     /* subtract two decimal numbers */
dec_t*n1;        /* input: minuend */
dec_t*n2;        /* input: subtrahend */
dec_t*n3;        /* output: difference */

int
decmul(*n1, *n2, *n3)     /* multiply two decimal numbers */
dec_t*n1;        /* input: multiplicand */
dec_t*n2;        /* input: multiplicand */
dec_t*n3;        /* output: product */

int
decdiv(*n1, *n2, *n3)    /* divide two decimal numbers */
dec_t*n1;        /* input: dividend */
dec_t*n2;        /* input: divisor */
dec_t*n3;        /* output: quotient */
```

## Description

These functions allow storage, conversion, and manipulation of packed decimal data on the BEA Tuxedo ATMI system. Note that the format in which the decimal data type is represented on the BEA Tuxedo ATMI system is different from its representation under CICS.

A thread in a multithreaded application may issue a call to any of the `decimal` conversion functions while running in any context state, including `TPINVALIDCONTEXT`.

## Native Decimal Representation

Decimals are represented on native BEA Tuxedo ATMI system nodes using the `dec_t` structure. This definition of this structure is as follows:

```
#define DECSIZE                 16
struct decimal {
       short dec_exp;           /* exponent base 100  */
       short dec_pos;           /* sign: 1=pos, 0=neg, -1=null */
       short dec_ndgts;         /* number of significant digits */
       char  dec_dgts[DECSIZE]; /* actual digits base 100 */
```

```
};
typedef struct decimal dec_t;
```

It should never be necessary for programmers to directly access the `dec_t` structure, but it is presented here nevertheless to give an understanding of the underlying data structure. If large amounts of decimal data need to be stored, the `stdecimal()` and `lddecimal()` functions may be used to obtain a more compact format. `dectoasc()`, `dectoint()`, `dectolong()`, `dectodbl()`, and `dectoflt()` allow the conversion of decimals to other data types. `deccvasc()`, `deccvint()`, `deccvlong()`, `deccvdbl()`, and `deccvflt()` allow the conversion of other data types to the decimal data type. `deccmp()` is the function which compares two decimals. It returns -1 if the first decimal is less than the second, 0 if the two decimals are equal, and 1 if the first decimal is greater than the second. A negative value other than -1 is returned if either of the arguments is invalid. `decadd()`, `decsub()`, `decmul()`, and `decdiv()` perform arithmetic operations on decimal numbers.

### Return Value

Unless otherwise stated, these functions return 0 on success and a negative value on error.

## getURLEntityCacheDir(3c)

### Name

`getURLEntityCacheDir()` - Specifies a Xerces class method for getting the absolute path to the location where the DTD, schema and Entity files are cached.

### Synopsis

```
char * getURLEntityCacheDir()
```

### Description

`getURLEntityCacheDir()` is a method that is called to find out the location where the DTD, schema and Entity files are cached. It returns the absolute path to the cached file location. This method is exclusively used in conjunction with the following two Xerces objects:

- XercesDOMParser
- SAXparser

## getURLEntityCaching(3c)

### Name

`GetURLEntityCaching()` - Specifies a Xerces class method for getting the caching mechanism for DTD, schema and Entity files.

### Synopsis

```
bool getURLEntityCaching()
```

### Description

`GetURLEntityCaching()` is a method that is called to find out if caching of the DTD, schema and Entity files are turned on or off. It returns *true* if caching is turned on and *false* if caching is turned off. This method is exclusively used in conjunction with the following two Xerces objects:

- XercesDOMParser
- SAXparser

## gp_mktime(3c)

### Name

`gp_mktime()`—Converts a `tm` structure to a calendar time.

### Synopsis

```
#include <time.h>
time_t gp_mktime (struct tm *timeptr);
```

### Description

`gp_mktime()` converts the time represented by the `tm` structure pointed to by *`timeptr`* into a calendar time (the number of seconds since 00:00:00 Universal Coordinated Time—UTC, January 1, 1970).

The `tm` structure has the following format:

```
struct tm {
  int tm_sec;     /* seconds after the minute [0, 61] */
  int tm_min;     /* minutes after the hour [0, 59] */
  int tm_hour;    /* hour since midnight [0, 23] */
  int tm_mday;    /* day of the month [1, 31] */
```

```
  int tm_mon;     /* months since January [0, 11] */
  int tm_year;    /* years since 1900 */
  int tm_wday;    /* days since Sunday [0, 6] */
  int tm_yday;    /* days since January 1 [0, 365] */
  int tm_isdst;   /* flag for daylight savings time */
};
```

In addition to computing the calendar time, `gp_mktime()` normalizes the supplied `tm` structure. The original values of the *`tm_wday`* and *`tm_yday`* components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated in the definition of the structure. On successful completion, the values of the *`tm_wday`* and *`tm_yday`* components are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to be within the appropriate ranges. The final value of *`tm_mday`* is not set until *`tm_mon`* and *`tm_year`* are determined.

The original values of the components may be either greater than or less than the specified range. For example, a *`tm_hour`* of -1 means 1 hour before midnight, *`tm_mday`* of 0 means the day preceding the current month, and *`tm_mon`* of -2 means 2 months before January of *`tm_year`*.

If *`tm_isdst`* is positive, the original values are assumed to be in the alternate time zone. If it turns out that the alternate time zone is not valid for the computed calendar time, then the components are adjusted to the main time zone. Likewise, if *`tm_isdst`* is zero, the original values are assumed to be in the main time zone and are converted to the alternate time zone if the main time zone is not valid. If *`tm_isdst`* is negative, the correct time zone is determined and the components are not adjusted.

Local time zone information is used as if `gp_mktime()` had called `tzset()`.

`gp_mktime()` returns the specified calendar time. If the calendar time cannot be represented, the function returns the value (`time_t`)-1.

A thread in a multithreaded application may issue a call to `gp_mktime()` while running in any context state, including `TPINVALIDCONTEXT`.

## Example

What day of the week is July 4, 2001?

```
#include <stdio.h>
#include <time.h>

static char *const wday[] = {
"Sunday", "Monday", "Tuesday", "Wednesday",
```

```
"Thursday", "Friday", "Saturday", "-unknown-"
};

struct tm time_str;
/*...*/
time_str.tm_year        = 2001 - 1900;
time_str.tm_mon         = 7 - 1;
time_str.tm_mday        = 4;
time_str.tm_hour        = 0;
time_str.tm_min         = 0;
time_str.tm_sec         = 1;
time_str.tm_isdst       = -1;
if (gp_mktime(time_str) == -1)
   time_str.tm_wday=7;
printf("%s\en", wday[time_str.tm_wday]);
```

## Notices

*tm_year* of the tm structure must be for year 1970 or later. Calendar times before 00:00:00 UTC, January 1, 1970 or after 03:14:07 UTC, January 19, 2038 cannot be represented.

## Portability

On systems where the C compilation system already provides the ANSI C `mktime()` function, `gp_mktime()` simply calls `mktime()` to do the conversion. Otherwise, the conversion is provided directly in `gp_mktime()`.

In the latter case, the TZ environment variable must be set. Note that in many installations, TZ is set to the correct value by default when the user logs on. The default value for TZ is GMT0. The format for TZ is the following:

```
stdoffset[dst[offset],[start[time],end[time]]]
```

*std* **and** *dst*

Three or more bytes that designate the standard time zone (*std*) and daylight savings time time zone (*dst*). Only *std* is required. If *dst* is missing, then daylight savings time does not apply in this locale. Uppercase and lowercase letters are allowed. Any characters except a leading colon (:), digits, a comma (,), a minus (-) or a plus (+) are allowed.

*offset*

Indicates the value one must add to the local time to arrive at Coordinated Universal Time. The *offset* has the following form: *hh*[:*mm*[:*ss*]]. The minutes (*mm*) and seconds (*ss*) are optional. The hour (*hh*) is required and may be a single digit. The *offset* following *std* is required. If no *offset* follows *dst*, daylight savings time is assumed to be one hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour must be between 0 and 24, and the minutes (and seconds) if present, between 0 and 59. Out of range values may cause unpredictable behavior. If preceded by a "-", the time zone is east of the Prime Meridian; otherwise it is west (which may be indicated by an optional preceding "+" sign).

*start*/*time*,*end*/*time*

Indicates when to change to and back from daylight savings time, where *start*/time describes when the change from standard time to daylight savings time occurs, and *end*/*time* describes when the change back happens. Each *time* field describes when, in current local time, the change is made.

The formats of *start* and *end* are one of the following:

J*n*

The Julian day *n* (1 *n* 365). Leap days are not counted. That is, in all years, February 28 is day 59 and March 1 is day 60. It is impossible to refer to the occasional February 29.

*n*

The zero-based Julian day (0 *n* 365). Leap days are counted, and it is possible to refer to February 29.

**M***m.n.d*

Day *d* (0 *d* 6) of week *n* of month *m* in the year (1 *n* 5, 1 *m* 12), where week 5 means "the last *d*-day in month *m*," which may occur in either the fourth or the fifth week). Week 1 is the first week in which day *d* occurs. Day 0 (zero) is Sunday.

Implementation specific defaults are used for *start* and *end* if these optional fields are not given.

The *time* has the same format as *offset* except that no leading sign ("-" or "+") is allowed. The default, if *time* is not specified, is 02:00:00.

### See Also

ctime(3c), getenv(3c), timezone(4) in a UNIX system reference manual

## nl_langinfo(3c)

### Name

nl_langinfo()—Language information.

### Synopsis

```
#include <nl_types.h>
#include <langinfo.h>

char *nl_langinfo (nl_item item);
```

### Description

nl_langinfo() returns a pointer to a NULL-terminated string containing information relevant to a particular language or cultural area defined in the programs locale. The manifest constant names and values of *item* are defined by langinfo.h.

For example:

```
nl_langinfo (ABDAY_1);
```

returns a pointer to the string "Dim" if the identified language is French and a French locale is correctly installed; or "Sun" if the identified language is English.

A thread in a multithreaded application may issue a call to nl_langinfo() while running in any context state, including TPINVALIDCONTEXT.

### Diagnostics

If setlocale() has not been called successfully, or if langinf0() data for a supported language is either not available or *item* is not defined therein, then nl_langinfo() returns a pointer to the corresponding string in the C locale. In all locales, nl_langinfo() returns a pointer to an empty string if *item* contains an invalid setting.

### Notices

The array pointed to by the return value should not be modified by the program. Subsequent calls to `nl_langinfo()` may overwrite the array.

### See Also

`setlocale(3c)`, `strftime(3c)`, `langinfo(5)`, `nl_types(5)`

## setlocale(3c)

### Name

`setlocale()`—Modifies and queries a program's locale.

### Synopsis

```
#include <locale.h>
char *setlocale (int category, const char *locale);
```

### Description

`setlocale()` selects the appropriate piece of the program's locale as specified by the *`category`* and *`locale`* arguments. The *`category`* argument may have the following values:

```
LC_CTYPE
LC_NUMERIC
LC_TIME
LC_COLLATE
LC_MONETARY
LC_MESSAGES
LC_ALL
```

These names are defined in the `locale.h` header file. For the BEA Tuxedo ATMI system compatibility functions, `setlocale()` allows only a single *`locale`* for all categories. Setting any category is treated the same as `LC_ALL`, which names the program's entire locale.

A value of "C" for *`locale`* specifies the default environment.

A value of "" for *`locale`* specifies that the locale should be taken from an environment variable. The environment variable `LANG` is checked for a locale.

At program startup, the equivalent of

```
setlocale(LC_ALL, "C")
```

is executed. This has the effect of initializing each category to the locale described by the environment "C".

If a pointer to a string is given for *locale*, setlocale() attempts to set the locale for all the categories to *locale*. The *locale* must be a simple locale, consisting of a single locale. If setlocale() fails to set the locale for any category, a NULL pointer is returned and the program's locale for all categories is not changed. Otherwise, locale is returned.

A NULL pointer for *locale* causes setlocale() to return the current locale associated with the *category*. The program's locale is not changed.

A thread in a multithreaded application may issue a call to setlocale() while running in any context state, including TPINVALIDCONTEXT.

### Files

```
$TUXDIR/locale/C/LANGINFO - time and money database for the C locale
$TUXDIR/locale/locale/* - locale specific information for each
locale $TUXDIR/locale/C/*_CAT - text messages for the C locale
```

### Note

A composite locale is not supported. A composite locale is a string beginning with a "/", followed by the locale of each category, separated by a "/".

### See Also

mklanginfo(1)

ctime(3c), ctype(3c), getdate(3c), localeconv(3c), strftime(3c), strtod(3c), printf(3S), environ(5) in a UNIX system reference manual

## setURLEntityCacheDir(3c)

### Name

setURLEntityCacheDir() - Specifies a Xerces class method for setting the directory where the DTD, schema and Entity files are to be cached.

### Synopsis

```
void setURLEntityCacheDir (const char* cachedir)
```

### Description

`setURLEntityCacheDir()` is method called when caching is turned on and you want the DTD, schema and Entity files to be cached to a specific directory. `cachedir` specifies the absolute path to the location of the files.

If this method is not called and caching is turned on either by calling the method `setURLEntityCaching()` or by not setting the environment variable, then the files are cached in the current directory. This method is exclusively used in conjunction with the following two Xerces objects:

- XercesDOMParser
- SAXparser

## setURLEntityCaching(3c)

### Name

`setURLEntityCaching()` - Specifies a Xerces class method for setting or unsetting DTD, schema or Entity file caching for the XML parser.

### Synopsis

```
void setURLEntityCaching (bool UseCache)
```

### Description

`setURLEntityCaching()`is a method that caches the DTD, schema and Entity files by default. It allows you to turn caching of the files on or off. `UseCache` is set to *false* if caching is to be turned off and set to *true* if caching is to be turned on. This method is exclusively used in conjunction with the following two Xerces objects:

- XercesDOMParser
- SAXparser

## strerror(3c)

### Name

`strerror()`—Gets error message string.

### Synopsis

```
#include <string.h>
char *strerror (int errnum);
```

### Description

`strerror` maps the error number in *`errnum`* to an error message string, and returns a pointer to that string. `strerror` uses the same set of error messages as `perror`. The returned string should not be overwritten.

A thread in a multithreaded application may issue a call to `strerror()` while running in any context state, including `TPINVALIDCONTEXT`.

### See Also

`perror`(3) in a UNIX system reference manual

## strftime(3c)

### Name

`strftime()`—Converts date and time to string.

### Synopsis

```
#include <time.h>

size_t *strftime (char *s, size_t maxsize, const char *format, const struct
tm *timeptr);
```

### Description

`strftime()` places characters into the array pointed to by *`s`* as controlled by the string pointed to by *`format`*. The *`format`* string consists of zero or more directives and ordinary characters. All ordinary characters (including the terminating NULL character) are copied unchanged into the array. For `strftime()`, no more than *`maxsize`* characters are placed into the array.

If *`format`* is (char *)0, then the locale's default format is used. The default format is the same as `"%c"`.

Each directive is replaced by appropriate characters as described in the following list. The appropriate characters are determined by the `LC_TIME` category of the program's locale and by the values contained in the structure pointed to by *`timeptr`*.

| Character | Description |
|---|---|
| %% | Same as % |
| %a | Locale’s abbreviated weekday name |
| %A | Locale’s full weekday name |
| %b | Locale’s abbreviated month name |
| %B | Locale’s full month name |
| %c | Locale’s appropriate date and time representation |
| %C | Locale’s date and time representation as produced by date(1) |
| %d | Day of month ( 01 - 31 ) |
| %D | Date as %m/%d/%y |
| %e | Day of month (1-31; single digits are preceded by a blank) |
| %h | Locale’s abbreviated month name. |
| %H | Hour ( 00 - 23 ) |
| %I | Hour ( 01 - 12 ) |
| %j | Day number of year ( 001 - 366 ) |
| %m | Month number ( 01 - 12 ) |
| %M | Minute ( 00 - 59 ) |
| %n | Same as \ |
| %p | Locale’s equivalent of either AM or PM |
| %r | Time as %I:%M:%S [AM\|PM] |
| %R | Time as %H:%M |
| %S | Seconds ( 00 - 61 ), allows for leap seconds |
| %t | Insert a tab |
| %T | Time as %H:%M:%S |

| | |
|---|---|
| %U | Week number of year ( 00 - 53 ), Sunday is the first day of week 1 |
| %w | Weekday number ( 0 - 6 ), Sunday = 0 |
| %W | Week number of year ( 00 - 53 ), Monday is the first day of week 1 |
| %x | Locale's appropriate date representation |
| %X | Locale's appropriate time representation |
| %y | Year within century ( 00 - 99 ) |
| %Y | Year as ccyy (for example, 1986) |
| %Z | Time zone name or no characters if no time zone exists |

The difference between %U and %W lies in which day is counted as the first of the week. Week number 01 is the first week in January starting with a Sunday for %U or a Monday for %W. Week number 00 contains those days before the first Sunday or Monday in January for %U and %W, respectively.

If the total number of resulting characters including the terminating NULL character is not more than *maxsize*, strftime(), returns the number of characters placed into the array pointed to by *s* not including the terminating NULL character. Otherwise, zero is returned and the contents of the array are indeterminate.

A thread in a multithreaded application may issue a call to strftime() while running in any context state, including TPINVALIDCONTEXT.

## Selecting the Output Language

By default, the output of strftime(), appears in U.S. English. The user can request that the output of strftime() be in a specific language by setting the *locale* for *category* LC_TIME in setlocale().

## Time Zone

The time zone is taken from the environment variable TZ. See ctime(3c) for a description of TZ.

## Examples

The example illustrates the use of strftime(). It shows what the string in str would look like if the structure pointed to by *tmptr* contains the values corresponding to Thursday, August 28, 1986 at 12:44:36 in New Jersey.

```
strftime (str, strsize, "%A %b %d %j", tmptr)
```

This results in `str` containing "Thursday Aug 28 240".

### Files

`$TUXDIR/locale/`*`locale`*`/LANGINFO`—file containing compiled locale-specific date and time information

### See Also

`mklanginfo(1)`, `setlocale(3c)`

## tpabort(3c)

### Name

`tpabort()`—Routine for aborting current transaction.

### Synopsis

```
#include <atmi.h>
int tpabort(long flags)
```

### Description

`tpabort()` signifies the abnormal end of a transaction. When this call returns, all changes made to resources during the transaction are undone. Like `tpcommit()`, this function can be called only by the initiator of a transaction. Participants (that is, service routines) can express their desire to have a transaction aborted by calling `tpreturn()` with `TPFAIL`.

If `tpabort()` is called while call descriptors exist for outstanding replies, then upon return from the function, the transaction is aborted and those descriptors associated with the caller's transaction are no longer valid. Call descriptors not associated with the caller's transaction remain valid.

For each open connection to a conversational server in transaction mode, `tpabort()` will send a `TPEV_DISCONIMM` event to the server, whether or not the server has control of a connection. Connections opened before `tpbegin()` or with the `TPNOTRAN` flag (that is, not in transaction mode) are not affected.

Currently, the sole argument to the `tpabort()` function, *`flags`*, is reserved for future use and should be set to 0.

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tpabort().

## Return Values

Upon failure, tpabort() returns -1 and sets tperrno to indicate the error condition.

## Errors

Upon failure, tpabort() sets tperrno to one of the following values:

**[**TPEINVAL**]**
*flags* is not equal to 0. The caller's transaction is not affected.

**[**TPEHEURISTIC**]**
Due to a heuristic decision, the work done on behalf of the transaction was partially committed and partially aborted.

**[**TPEHAZARD**]**
Due to some failure, the work done on behalf of the transaction could have been heuristically completed.

**[**TPEPROTO**]**
tpabort() was called improperly (for example, by a participant).

**[**TPESYSTEM**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPEOS**]**
An operating system error has occurred.

## Notices

When using tpbegin(), tpcommit(), and tpabort() to delineate a BEA Tuxedo ATMI system transaction, it is important to remember that only the work done by a resource manager that meets the XA interface (and is linked to the caller appropriately) has transactional properties. All other operations performed in a transaction are not affected by either tpcommit() or tpabort().

## See Also

tpbegin(3c), tpcommit(3c), tpgetlev(3c)

## tpacall(3c)

### Name

`tpacall()`—Routine for sending a service request.

### Synopsis

```
#include <atmi.h>
int tpacall(char *svc, char *data, long len, long flags)
```

### Description

`tpacall()` sends a request message to the service named by *`svc`*. The request is sent out at the priority defined for *`svc`* unless overridden by a previous call to `tpspri()`. If *`data`* is non-NULL, it must point to a buffer previously allocated by `tpalloc()` and *`len`* should specify the amount of data in the buffer that should be sent. Note that if *`data`* points to a buffer of a type that does not require a length to be specified, (for example, an `FML` fielded buffer), then *`len`* is ignored (and may be 0). If *`data`* is NULL, *`len`* is ignored and a request is sent with no data portion. The type and subtype of *`data`* must match one of the types and subtypes recognized by *`svc`*. Note that for each request sent while in transaction mode, a corresponding reply must ultimately be received.

The following is a list of valid *`flags`*:

`TPNOTRAN`
: If the caller is in transaction mode and this flag is set, then when *`svc`* is invoked, it is not performed on behalf of the caller's transaction. If *`svc`* belongs to a server that does not support transactions, then this flag must be set when the caller is in transaction mode. Note that *`svc`* may still be invoked in transaction mode but it will not be the same transaction: a *`svc`* may have as a configuration attribute that it is automatically invoked in transaction mode. A caller in transaction mode that sets this flag is still subject to the transaction timeout (and no other). If a service fails that was invoked with this flag, the caller's transaction is not affected.

`TPNOREPLY`
: Informs `tpacall()` that a reply is not expected. When `TPNOREPLY` is set, the function returns 0 on success, where 0 is an invalid descriptor. When the caller is in transaction mode, this setting cannot be used unless `TPNOTRAN` is also set.

`TPNOBLOCK`
: The request is not sent if a blocking condition exists (for example, the internal buffers into which the message is transferred are full). When `TPNOBLOCK` is not specified and a

blocking condition exists, the caller blocks until the condition subsides or a timeout occurs (either transaction or blocking timeout).

TPNOTIME
: This flag signifies that the caller is willing to block indefinitely and wants to be immune to blocking timeouts. Transaction timeouts may still occur.

TPSIGRSTRT
: If a signal interrupts any underlying system calls, then the interrupted system call is reissued.

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tpacall().

## Return Values

Upon successful completion, tpacall() returns a descriptor that can be used to receive the reply of the request sent.

Upon failure, tpacall() returns a value of -1 and sets tperrno to indicate the error condition.

## Errors

Upon failure, tpacall() sets tperrno to one of the following values. (Unless otherwise noted, failure does not affect the caller's transaction, if one exists.)

**[**TPEINVAL**]**
: Invalid arguments were given (for example, *svc* is NULL, *data* does not point to space allocated with tpalloc(), or *flags* are invalid).

**[**TPENOENT**]**
: Cannot send to *svc* because it does not exist or is a conversational service.

**[**TPEITYPE**]**
: The type and subtype of *data* is not one of the allowed types and subtypes that *svc* accepts.

**[**TPELIMIT**]**
: The caller's request was not sent because the maximum number of outstanding asynchronous requests has been reached.

**[**TPETRAN**]**
: *svc* belongs to a server that does not support transactions and TPNOTRAN was not set.

**[**TPETIME**]**

This error code indicates that either a timeout has occurred or tpacall() has been attempted, in spite of the fact that the current transaction is already marked rollback only.

If the caller is in transaction mode, then either the transaction is already rollback only or a transaction timeout has occurred. The transaction is marked abort-only. If the caller is not in transaction mode, a blocking timeout has occurred. (A blocking timeout cannot occur if TPNOBLOCK and/or TPNOTIME is specified.)

If a transaction timeout has occurred, then, with one exception, any attempts to send new requests or receive outstanding replies will fail with TPETIME until the transaction has been aborted. The exception is a request that does not block, expects no reply, and is not sent on behalf of the caller's transaction (that is, tpacall() with TPNOTRAN, TPNOBLOCK, and TPNOREPLY set).

When a service fails inside a transaction, the transaction is put into the TX_ROLLBACK_ONLY state. This state is treated, for most purposes, as though it were equivalent to a timeout. All further ATMI calls for this transaction (with the exception of those issued in the circumstances described in the previous paragraph) will fail with TPETIME.

**[**TPEBLOCK**]**

A blocking condition exists and TPNOBLOCK was specified.

**[**TPGOTSIG**]**

A signal was received and TPSIGRSTRT was not specified.

**[**TPEPROTO**]**

tpacall() was called improperly.

**[**TPESYSTEM**]**

A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPEOS**]**

An operating system error has occurred. If a message queue on a remote location is filled, TPEOS may be returned even if tpacall() returned successfully.

## See Also

tpalloc(3c), tpcall(3c), tpcancel(3c), tpgetrply(3c), tpgprio(3c), tpsprio(3c)

## tpadmcall(3c)

### Name

`tpadmcall()`—Administers unbooted application.

### Synopsis

```
#include <atmi.h>
#include <fml32.h>
#include <tpadm.h>

int tpadmcall(FBFR32 *inbuf, FBFR32 **outbuf, long flags)
```

### Description

`tpadmcall()` is used to retrieve and update attributes of an unbooted application. It may also be used in an active application to perform direct retrievals of a limited set of attributes without requiring communication to an external process. This function provides sufficient capability such that complete system configuration and administration can take place through system provided interface routines.

*`inbuf`* is a pointer to an FML32 buffer previously allocated with `tpalloc()` that contains the desired administrative operation and its parameters.

*`outbuf`* is the address of a pointer to the FML32 buffer that should contain the results. *`outbuf`* must point to an FML32 buffer originally allocated by `tpalloc()`. If the same buffer is to be used for both sending and receiving, *`outbuf`* should be set to the address of *`inbuf`*.

Currently, `tpadmcall()`'s last argument, *`flags`*, is reserved for future use and must be set to 0.

`MIB(5)` should be consulted for generic information on construction of administrative requests. `TM_MIB(5)` and `APPQ_MIB(5)` should be consulted for information on the classes that are accessible through `tpadmcall()`.

There are four modes in which calls to `tpadmcall()` can be made.

**Mode 1: Unbooted, Unconfigured Application:**

The caller is assumed to be the administrator of the application. The only operations permitted are to SET a NEW T_DOMAIN class object, thus defining an initial configuration for the application, and to GET and SET objects of the classes defined in `APPQ_MIB()`.

**Mode 2: Unbooted, Configured Application:**

The caller is assigned administrator or other privileges based on a comparison of their UID/GID to that defined in the configuration for the administrator on the local system. The caller may GET and SET any attributes for any class in `TM_MIB()` and `APPQ_MIB()` for which they have the appropriate permissions. Note that some classes contain only attributes that are inaccessible in an unbooted application and attempts to access these classes will fail.

**Mode 3: Booted Application, Unattached Process:**

The caller is assigned administrator or other privileges based on a comparison of their UID/GID to that defined in the configuration for the administrator on the local system. The caller may GET any attributes for any class in `TM_MIB()` for which they have the appropriate permissions. Similarly, the caller may GET and SET any attributes for any class in `APPQ_MIB()`, subject to class-specific restrictions. Attributes accessible only while ACTIVE will not be returned.

**Mode 4: Booted Application, Attached Process:**

Permissions are determined from the authentication key assigned at `tpinit()` time. The caller may GET any attributes for any class in `TM_MIB()` for which they have the appropriate permissions. Additionally, the caller may GET and SET any attributes for any class in `APPQ_MIB()`, subject to class-specific restrictions.

Access to and update of binary BEA Tuxedo ATMI system application configuration files through this interface routine is controlled through the use of UNIX system permissions on directory names and filenames.

In a multithreaded application, a thread in the `TPINVALIDCONTEXT` state is not allowed to issue a call to `tpadmcall()`.

## Environment Variables

The following environment variables must be set prior to calling this routine:

`TUXCONFIG`

Name of the file or device on which the binary BEA Tuxedo system configuration file for this application is or should be stored.

## Notices

Use of the `TA_OCCURS` attribute on `GET` requests is not supported when using `tpadmcall()`. `GETNEXT` requests are not supported when using `tpadmcall()`.

## Return Values

`tpadmcall()` returns 0 on success and -1 on failure.

## Errors

Upon failure, tpadmcall() sets tperrno to one of the following values:

**Note:** Except for TPEINVAL, the caller's output buffer, *outbuf*, will be modified to include TA_ERROR, TA_STATUS, and possibly TA_BADFLD attributes to further qualify the error condition. See MIB(5), TM_MIB(5), and APPQ_MIB(5) for an explanation of possible error codes returned in this fashion.

**[**TPEINVAL**]**
Invalid arguments were specified. The *flags* value is invalid or *inbuf* or *outbuf* are not pointers to typed buffers of type "FML32."

**[**TPEMIB**]**
The administrative request failed. *outbuf* is updated and returned to the caller with FML32 fields indicating the cause of the error as is discussed in MIB(5) and TM_MIB(5).

**[**TPEPROTO**]**
tpadmcall() was called improperly.

**[**TPERELEASE**]**
tpadmcall() was called with the TUXCONFIG environment variable pointing to a different release version configuration file.

**[**TPEOS**]**
An operating system error has occurred. A numeric value representing the system call that failed is available in Uunixerr.

**[**TPESYSTEM**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to userlog().

## Interoperability

This interface supports access and update to the local configuration file and bulletin board only; therefore, there are no interoperability concerns.

## Portability

This interface is available only on UNIX system sites running BEA Tuxedo ATMI release 5.0 or later.

### Files

The following library files are required:

`${TUXDIR}/lib/libtmib.a`, `${TUXDIR}/lib/libqm.a`,
`${TUXDIR}/lib/libtmib.so.`*`<rel>`*, `${TUXDIR}/lib/libqm.so.`*`<rel>`*,
`${TUXDIR}/lib/libtmib.lib`, `${TUXDIR}/lib/libqm.lib`

The libraries must be linked manually when using buildclient. The user must use:
`-L${TUXDIR}/lib -ltmid -lqm`

### See Also

`ACL_MIB(5)`, `APPQ_MIB(5)`, `EVENT_MIB(5)`, `MIB(5)`, `TM_MIB(5)`, `WS_MIB(5)`
*Setting Up a BEA Tuxedo Application*
*Administering a BEA Tuxedo Application at Run Time*

## tpadvertise(3c)

### Name

`tpadvertise()`—Routine for advertising a service name.

### Synopsis

```
#include <atmi.h>
int tpadvertise(char *svcname, void (*func)(TPSVCINFO *))
```

### Description

`tpadvertise()` allows a server to advertise the services that it offers. By default, a server's services are advertised when it is booted and unadvertised when it is shutdown.

All servers belonging to a Multiple Server, Single Queue (MSSQ) set must offer the same set of services. These routines enforce this rule by affecting the advertisements of all servers sharing an MSSQ set.

`tpadvertise()` advertises *`svcname`* for the server (or the set of servers sharing the caller's MSSQ set). *`svcname`* should be 15 characters or less, but cannot be NULL or the NULL string (""). (See *SERVICES section of `UBBCONFIG(5)`.)*`func`* is the address of a BEA Tuxedo ATMI system service function. This function will be invoked whenever a request for *`svcname`* is received by the server. *`func`* cannot be NULL. Explicitly specified function names (see `servopts(5)`) can be up to 128 characters long. Names longer than 15 characters are accepted

and truncated to 15 characters. Users should make sure that truncated names do not match other service names.

If *svcname* is already advertised for the server and *func* matches its current function, then tpadvertise() returns success (this includes truncated names that match already advertised names). However, if *svcname* is already advertised for the server but *func* does not match its current function, then an error is returned (this can happen if truncated names match already advertised names).

Service names starting with dot (.) are reserved for administrative services. An error will be returned if an application attempts to advertise one of these services.

## Return Values

Upon failure, tpadvertise() returns -1 and sets tperrno to indicate the error condition.

## Errors

Upon failure, tpadvertise() sets tperrno to one of the following values:

**[**TPEINVAL**]**
*svcname* is NULL or the NULL string (“”),or begins with a “.” or *func* is NULL.

**[**TPELIMIT**]**
*svcname* cannot be advertised because of space limitations. (See MAXSERVICES in the RESOURCES section of UBBCONFIG(5).)

**[**TPEMATCH**]**
*svcname* is already advertised for the server but with a function other than *func*. Although the function fails, *svcname* remains advertised with its current function (that is, *func* does not replace the current function).

**[**TPEPROTO**]**
tpadvertise() was called in an improper context (for example, by a client).

**[**TPESYSTEM**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPEOS**]**
An operating system error has occurred.

## See Also

tpservice(3c), tpunadvertise(3c)

# tpalloc(3c)

## Name

tpalloc()—Routine for allocating typed buffers.

## Synopsis

```
#include <atmi.h>
char * tpalloc(char *type, char *subtype, long size)
```

## Description

tpalloc() returns a pointer to a buffer of type *type*. Depending on the type of buffer, both *subtype* and *size* are optional. The BEA Tuxedo ATMI system provides a variety of typed buffers, and applications are free to add their own buffer types. Consult tuxtypes(5) for more details.

If *subtype* is non-NULL in tmtype_sw for a particular buffer type, then *subtype* must be specified when tpalloc() is called. The allocated buffer will be at least as large as the larger of *size* and dfltsize, where dfltsize is the default buffer size specified in tmtype_sw for the particular buffer type. For buffer type STRING the minimum is 512 bytes; for buffer types FML and VIEW the minimum is 1024 bytes.

Note that only the first eight bytes of *type* and the first 16 bytes of *subtype* are significant.

Because some buffer types require initialization before they can be used, tpalloc() initializes a buffer (in a BEA Tuxedo ATMI system-specific manner) after it is allocated and before it is returned. Thus, the buffer returned to the caller is ready for use. Note that unless the initialization routine cleared the buffer, the buffer is not initialized to zeros by tpalloc().

A thread in a multithreaded application may issue a call to tpalloc() while running in any context state, including TPINVALIDCONTEXT.

## Return Values

Upon successful completion, tpalloc() returns a pointer to a buffer of the appropriate type aligned on a long word; otherwise, it returns NULL and sets tperrno to indicate the condition.

## Errors

Upon failure, tpalloc() sets tperrno to one of the following values:

**[**TPEINVAL**]**
Invalid arguments were given (for example, *type* is NULL).

**[**TPENOENT**]**
No entry in tmtype_sw matches *type* and, if non-NULL, *subtype*.

**[**TPEPROTO**]**
tpalloc() was called improperly.

**[**TPESYSTEM**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPEOS**]**
An operating system error has occurred.

### Usage

If buffer initialization fails, the allocated buffer is freed and tpalloc() fails returning NULL.

This function should not be used in concert with malloc(), realloc(), or free() in the C library (for example, a buffer allocated with tpalloc() should not be freed with free()).

Two buffer types are supported by any compliant implementation of the BEA Tuxedo ATMI system extension. Details are in the Introduction to the C Language Application-to-Transaction Monitor Interface.

### See Also

tpfree(3c), tprealloc(3c), tptypes(3c)

## tpbegin(3c)

### Name

tpbegin()—Routine for beginning a transaction.

### Synopsis

```
#include <atmi.h>
int tpbegin(unsigned long timeout, long flags)
```

### Description

A transaction in the BEA Tuxedo ATMI system is used to define a single logical unit of work that either wholly succeeds or has no effect whatsoever. A transaction allows work being performed in many processes, at possibly different sites, to be treated as an atomic unit of work. The initiator of a transaction uses tpbegin() and either tpcommit() or tpabort() to delineate the

operations within a transaction. Once tpbegin() is called, communication with any other program can place the latter (of necessity, a server) in "transaction mode" (that is, the server's work becomes part of the transaction). Programs that join a transaction are called participants. A transaction always has one initiator and can have several participants. Only the initiator of a transaction can call tpcommit() or tpabort(). Participants can influence the outcome of a transaction by the return values (*rval*s) they use when they call tpreturn(). Once in transaction mode, any service requests made to servers are processed on behalf of the transaction (unless the requester explicitly specifies otherwise).

Note that if a program starts a transaction while it has any open connections that it initiated to conversational servers, these connections will not be upgraded to transaction mode. It is as if the TPNOTRAN flag had been specified on the tpconnect() call.

tpbegin()'s first argument, *timeout*, specifies that the transaction should be allowed at least *timeout* seconds before timing out. Once a transaction times out it must be marked abort-only. If *timeout* is 0, then the transaction is given the maximum number of seconds allowed by the system before timing out (that is, the timeout value equals the maximum value for an unsigned long as defined by the system).

Currently, tpbegin()'s second argument, *flags*, is reserved for future use and must be set to 0.

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tpbegin().

## Return Values

Upon failure, tpbegin() returns -1 and sets tperrno to indicate the error condition.

## Errors

Upon failure, tpbegin() sets tperrno to one of the following values:

**[**TPEINVAL**]**
*flags* is not equal to 0.

**[**TPETRAN**]**
The caller cannot be placed in transaction mode because an error occurred starting the transaction.

**[**TPEPROTO**]**
tpbegin() was called in an improper context (for example, the caller is already in transaction mode).

**[**TPESYSTEM**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPEOS**]**
An operating system error has occurred.

### Notices

When using tpbegin(), tpcommit(), and tpabort() to delineate a BEA Tuxedo ATMI system transaction, it is important to remember that only the work done by a resource manager that meets the XA interface (and is linked to the caller appropriately) has transactional properties. All other operations performed in a transaction are not affected by either tpcommit() or tpabort(). See buildserver() for details on linking resource managers that meet the XA interface into a server such that operations performed by that resource manager are part of a BEA Tuxedo ATMI system transaction.

### See Also

tpabort(3c), tpcommit(3c), tpgetlev(3c), tpscmt(3c)

## tpbroadcast(3c)

### Name

tpbroadcast()—Routine to broadcast notification by name.

### Synopsis

```
#include <atmi.h>

int tpbroadcast(char *lmid, char *usrname, char *cltname,
  char *data,  long len, long flags)
```

### Description

tpbroadcast() allows a client or server to send unsolicited messages to registered clients within the system. The target client set consists of those clients matching identifiers passed to tpbroadcast(). Wildcards can be used in specifying identifiers.

*lmid*, *usrname*, and *cltname* are logical identifiers used to select the target client set. A NULL value for any argument constitutes a wildcard for that argument. A wildcard argument matches all client identifiers for that field. A 0-length string for any argument matches only 0-length client

identifiers. Each identifier must meet the size restrictions defined for the system to be considered valid, that is, each identifier must be between 0 and MAXTIDENT characters in length.

The data portion of the request is pointed to by *data*, a buffer previously allocated by tpalloc(). *len* specifies how much of *data* to send. Note that if *data* points to a buffer type that does not require a length to be specified (for example, an FML fielded buffer), then *len* is ignored (and may be 0). Also, *data* may be NULL, in which case *len* is ignored. The buffer passes through the typed buffer switch routines just as any other outgoing or incoming message would; for example, encode/decode are performed automatically.

The following is a list of valid *flags*:

TPNOBLOCK
: The request is not sent if a blocking condition exists (for example, the internal buffers into which the message is transferred are full).

TPNOTIME
: This flag signifies that the caller is willing to block indefinitely and wants to be immune to blocking timeouts. Transaction timeouts may still occur.

TPSIGRSTRT
: If a signal interrupts any underlying system calls, then the interrupted system call is reissued. Upon successful return from tpbroadcast(), the message has been delivered to the system for forwarding to the selected clients. tpbroadcast() does not wait for the message to be delivered to each selected client.

  In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tpbroadcast().

## Return Values

Upon failure, tpbroadcast() returns -1 and sets tperrno to indicate the error condition.

## Errors

Upon failure, tpbroadcast()sends no broadcast messages to application clients and sets tperrno to one of the following values:

**[**TPEINVAL**]**
: Invalid arguments were given (for example, identifiers too long or invalid flags). Note that use of an illegal LMID will cause tpbroadcast() to fail and return TPEINVAL. However, non-existent user or client names will simply successfully broadcast to no one.

**[**TPETIME**]**
A blocking timeout occurred. (A blocking timeout cannot occur if TPNOBLOCK and/or TPNOTIME is specified.)

**[**TPEBLOCK**]**
A blocking condition was found on the call and TPNOBLOCK was specified.

**[**TPGOTSIG**]**
A signal was received and TPSIGRSTRT was not specified.

**[**TPEPROTO**]**
tpbroadcast() was called improperly.

**[**TPESYSTEM**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPEOS**]**
An operating system error has occurred.

## Portability

The interfaces described in tpnotify(3c) are supported on native site UNIX-based processors. In addition, the routines tpbroadcast() and tpchkunsol() as well as the function tpsetunsol() are supported on UNIX and MS-DOS workstation processors.

## Usage

Clients that select signal-based notification may not be signal-able by the system due to signal restrictions. When this occurs, the system generates a log message that it is switching notification for the selected client to dip-in and the client is notified then and thereafter via dip-in notification. (See the description of the RESOURCES NOTIFY parameter in UBBCONFIG() for a detailed discussion of notification methods.)

Because signaling of clients is always done by the system, the behavior of notification is always consistent, regardless of where the originating notification call is made. Therefore to use signal-based notification:

- A native client must be running as an application administrator
- A Workstation client is not required to be running as the application administrator

The ID for the application administrator is identified as part of the configuration for the application.

If signal-based notification is selected for a client, then certain ATMI calls can fail, returning TPGOTSIG due to receipt of an unsolicited message if TPSIGRSTRT is not specified. See UBBCONFIG(5) and tpinit(3c) for more information on notification method selection.

### See Also

tpalloc(3c), tpinit(3c), tpnotify(3c), tpterm(3c), UBBCONFIG(5)

## tpcall(3c)

### Name

tpcall()—Routine for sending service request and awaiting its reply.

### Synopsis

```
int tpcall(char *svc, char *idata, long ilen, char **odata, long \
   *olen, long flags)
```

### Description

tpcall() sends a request and synchronously awaits its reply. A call to this function is the same as calling tpacall() immediately followed by tpgetrply(). tpcall() sends a request to the service named by *svc*. The request is sent out at the priority defined for *svc* unless overridden by a previous call to tpspri(). The data portion of a request is pointed to by *idata*, a buffer previously allocated by tpalloc(). *ilen* specifies how much of *idata* to send. Note that if *idata* points to a buffer of a type that does not require a length to be specified, (for example, an FML fielded buffer), then *ilen* is ignored (and may be 0). Also, *idata* may be NULL, in which case *ilen* is ignored. The type and subtype of *idata* must match one of the types and subtypes recognized by *svc*.

*odata* is the address of a pointer to the buffer where a reply is read into, and *olen* points to the length of that reply. **odata* must point to a buffer originally allocated by tpalloc(). If the same buffer is to be used for both sending and receiving, *odata* should be set to the address of *idata*. FML and FML32 buffers often assume a minimum size of 4096 bytes; if the reply is larger than 4096, the size of the buffer is increased to a size large enough to accommodate the data being returned. Also, if *idata* and **odata* were equal when tpcall() was invoked, and **odata* is changed, then *idata* no longer points to a valid address. Using the old address can lead to data corruption or process exceptions. As of release 6.4, the default allocation for buffers is 1024 bytes. Also, historical information is maintained on recently used buffers, allowing a buffer of optimal size to be reused as a return buffer.

Buffers on the sending side that may be only partially filled (for example, FML or STRING buffers) will have only the amount that is used send. The system may then enlarge the received data size by some arbitrary amount. This means that the receiver may receive a buffer that is smaller than what was originally allocated by the sender, yet larger than the data that was sent.

The receive buffer may grow, or it may shrink, and its address almost invariably changes, as the system swaps buffers around internally. To determine whether (and how much) a reply buffer changed in size, compare its total size before `tpgetrply()` was issued with *`len`. See "Introduction to the C Language Application-to-Transaction Monitor Interface" for more information about buffer management.

If *`olen` is 0 upon return, then the reply has no data portion and neither *`odata` nor the buffer it points to were modified. It is an error for *`odata` or `olen` to be NULL.

The following is a list of valid `flags`:

`TPNOTRAN`
: If the caller is in transaction mode and this flag is set, then when `svc` is invoked, it is not performed on behalf of the caller's transaction. Note that `svc` may still be invoked in transaction mode but it will not be the same transaction: a `svc` may have as a configuration attribute that it is automatically invoked in transaction mode. A caller in transaction mode that sets this flag is still subject to the transaction timeout (and no other). If a service fails that was invoked with this flag, the caller's transaction is not affected.

`TPNOCHANGE`
: By default, if a buffer is received that differs in type from the buffer pointed to by *`odata`, then *`odata`'s buffer type changes to the received buffer's type so long as the receiver recognizes the incoming buffer type. When this flag is set, the type of the buffer pointed to by *`odata` is not allowed to change. That is, the type and subtype of the received buffer must match the type and subtype of the buffer pointed to by *`odata`.

`TPNOBLOCK`
: The request is not sent if a blocking condition exists (for example, the internal buffers into which the message is transferred are full). Note that this flag applies only to the send portion of `tpcall()`: the function may block waiting for the reply. When `TPNOBLOCK` is not specified and a blocking condition exists, the caller blocks until the condition subsides or a timeout occurs (either transaction or blocking timeout).

`TPNOTIME`
: This flag signifies that the caller is willing to block indefinitely and wants to be immune to blocking timeouts. However, if the caller is in transaction mode, this flag has no effect; it is subject to the transaction timeout limit. Transaction timeouts may still occur.

TPSIGRSTRT
: If a signal interrupts any underlying system calls, then the interrupted system call is reissued.

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tpcall().

## Return Values

Upon successful return from tpcall() or upon return where tperrno is set to TPESVCFAIL, tpurcode() contains an application defined value that was sent as part of tpreturn().

Upon failure, tpcall() returns -1 and sets tperrno to indicate the error condition. If a call fails with a particular tperrno value, a subsequent call to tperrordetail(), with no intermediate ATMI calls, may provide more detailed information about the generated error. Refer to the tperrordetail(3c) reference page for more information.

## Errors

Upon failure, tpcall() sets tperrno to one of the following values. (Unless otherwise noted, failure does not affect the caller's transaction, if one exists.)

**[**TPEINVAL**]**
: Invalid arguments were given (for example, *svc* is NULL or *flags* are invalid).

**[**TPENOENT**]**
: Cannot send to *svc* because it does not exist, or it is a conversational service, or the name provided begins with a dot (.).

**[**TPEITYPE**]**
: The type and subtype of *idata* is not one of the allowed types and subtypes that *svc* accepts.

**[**TPEOTYPE**]**
: Either the type and subtype of the reply are not known to the caller; or, TPNOCHANGE was set in *flags* and the type and subtype of **odata* do not match the type and subtype of the reply sent by the service. Neither **odata*, its contents, nor **olen* is changed. If the service request was made on behalf of the caller's current transaction, then the transaction is marked abort-only since the reply is discarded.

**[**TPETRAN**]**
: *svc* belongs to a server that does not support transactions and TPNOTRAN was not set.

[TPETIME]

This error code indicates that either a timeout has occurred or tpcall() has been attempted, in spite of the fact that the current transaction is already marked rollback only.

If the caller is in transaction mode, then either the transaction is already rollback only or a transaction timeout has occurred. The transaction is marked abort-only. If the caller is not in transaction mode, a blocking timeout has occurred. (A blocking timeout cannot occur if TPNOBLOCK and/or TPNOTIME is specified.) In either case, no changes are made to *odata, its contents, or *olen.

If a transaction timeout has occurred, then, with one exception, any attempts to send new requests or receive outstanding replies will fail with TPETIME until the transaction has been aborted. The exception is a request that does not block, expects no reply, and is not sent on behalf of the caller's transaction (that is, tpacall() with TPNOTRAN, TPNOBLOCK, and TPNOREPLY set).

When a service fails inside a transaction, the transaction is put into the TX_ROLLBACK_ONLY state. This state is treated, for most purposes, as though it were equivalent to a timeout. All further ATMI calls for this transaction (with the exception of those issued in the circumstances described in the previous paragraph) will fail with TPETIME.

[TPESVCFAIL]

The service routine sending the caller's reply called tpreturn() with TPFAIL. This is an application-level failure. The contents of the service's reply, if one was sent, is available in the buffer pointed to by *_odata_. If the service request was made on behalf of the caller's current transaction, then the transaction is marked abort-only. Note that regardless of whether the transaction has timed out, the only valid communications before the transaction is aborted are calls to tpacall() with TPNOREPLY, TPNOTRAN, and TPNOBLOCK set.

[TPESVCERR]

A service routine encountered an error either in tpreturn(3c) or tpforward(3c) (for example, bad arguments were passed). No reply data is returned when this error occurs (that is, neither *_odata_, its contents, nor *_olen_ is changed). If the service request was made on behalf of the caller's transaction (that is, TPNOTRAN was not set), then the transaction is marked abort-only. Note that regardless of whether the transaction has timed out, the only valid communications before the transaction is aborted are calls to tpacall() with TPNOREPLY, TPNOTRAN, and TPNOBLOCK set. If either SVCTIMEOUT in the UBBCONFIG file or TA_SVCTIMEOUT in the TM_MIB is non-zero, TPESVCERR is returned when a service timeout occurs.

**[**TPEBLOCK**]**
A blocking condition was found on the send call and TPNOBLOCK was specified.

**[**TPGOTSIG**]**
A signal was received and TPSIGRSTRT was not specified.

**[**TPEPROTO**]**
tpcall() was called improperly.

**[**TPESYSTEM**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPEOS**]**
An operating system error has occurred. If a message queue on a remote location is filled, TPEOS may be returned even if tpcall() returned successfully.

### See Also

tpacall(3c),tpalloc(3c),tperrordetail(3c),tpforward(3c),tpfree(3c), tpgprio(3c),tprealloc(3c),tpreturn(3c),tpsprio(3c),tpstrerrordetail(3c), tptypes(3c)

## tpcancel(3c)

### Name

tpcancel()—Routine for canceling a call descriptor for outstanding reply.

### Synopsis

```
#include <atmi.h>
int tpcancel(int cd)
```

### Description

tpcancel() cancels a call descriptor, *cd*, returned by tpacall(). It is an error to attempt to cancel a call descriptor associated with a transaction.

Upon success, *cd* is no longer valid and any reply received on behalf of *cd* will be silently discarded.

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tpcancel().

### Return Values

Upon failure, `tpcancel()` returns -1 and sets `tperrno` to indicate the error condition.

### Errors

Upon failure, `tpcancel()` sets `tperrno` to one of the following values:

**[**`TPEBADDESC`**]**
*cd* is an invalid descriptor.

**[**`TPETRAN`**]**
*cd*() is associated with the caller's transaction. *cd* remains valid and the caller's current transaction is not affected.

**[**`TPEPROTO`**]**
`tpcancel()` was called improperly.

**[**`TPESYSTEM`**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**`TPEOS`**]**
An operating system error has occurred.

### See Also

`tpacall(3c)`

## tpchkauth(3c)

### Name

`tpchkauth()`—Routine for checking if authentication required to join an application.

### Synopsis

```
#include <atmi.h>

int tpchkauth(void)
```

### Description

`tpchkauth()` checks if authentication is required by the application configuration. This is typically used by application clients prior to calling `tpinit()` to determine if a password should be obtained from the user.

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tpchkauth().

## Return Values

Upon success, tpchkauth() returns one of the following non-negative values:

TPNOAUTH
: Indicates that no authentication is required.

TPSYSAUTH
: Indicates that system authentication only is required.

TPAPPAUTH
: Indicates that both system and application specific authentication are required.

Upon failure, tpchkauth() returns -1 and sets tperrno to indicate the error condition.

## Errors

Upon failure, tpchkauth() sets tperrno to one of the following values:

**[**TPESYSTEM**]**
: A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPEOS**]**
: An operating system error has occurred.

## Interoperability

tpchkauth() is available only on sites running release 4.2 or later.

### Portability

The interfaces described in tpchkauth(3c) are supported on UNIX, Windows, and MS-DOS operating systems.

### See Also

tpinit(3c)

## tpchkunsol(3c)

### Name

tpchkunsol()—Routine for checking for unsolicited message.

### Synopsis

```
#include <atmi.h>
int tpchkunsol(void)
```

### Description

tpchkunsol() is used by a client to trigger checking for unsolicited messages. Calls to this routine in a client using signal-based notification do nothing and return immediately. This call has no arguments. Calls to this routine can result in calls to an application-defined unsolicited message handling routine by the BEA Tuxedo ATMI system libraries.

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tpchkunsol().

### Return Values

Upon successful completion, tpchkunsol() returns the number of unsolicited messages dispatched; otherwise it returns -1 and sets tperrno to indicate the error condition.

### Errors

Upon failure, tpchkunsol() sets tperrno to one of the following values:

**[**TPEPROTO**]**
tpchkunsol() was called in an improper context (for example, from within a server).

**[**TPESYSTEM**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPEOS**]**
An operating system error has occurred.

### Portability

The interfaces described in tpnotify(3c) are supported on native site UNIX-based processors. In addition, the routines tpbroadcast() and tpchkunsol() as well as the function tpsetunsol() are supported on UNIX and MS-DOS workstation processors.

Clients that select signal-based notification may not be signal-able by the system due to signal restrictions. When this occurs, the system generates a log message that it is switching notification for the selected client to dip-in and the client is notified then and thereafter via dip-in notification. (See the description of the RESOURCES NOTIFY parameter in UBBCONFIG(5) for a detailed discussion of notification methods.)

Because signaling of clients is always done by the system, the behavior of notification is always consistent, regardless of where the originating notification call is made. Therefore to use signal-based notification:

- A native client must be running as an application administrator
- A Workstation client is not required to be running as the application administrator

The ID for the application administrator is identified as part of the configuration for the application.

If signal-based notification is selected for a client, then certain ATMI calls can fail, returning TPGOTSIG due to receipt of an unsolicited message if TPSIGRSTRT is not specified. See UBBCONFIG(5) and tpinit(3c) for more information on notification method selection.

### See Also

tpbroadcast(3c),tpinit(3c),tpnotify(3c),tpsetunsol(3c)

## tpclose(3c)

### Name

tpclose()—Routine for closing a resource manager.

### Synopsis

```
#include <atmi.h>
int tpclose(void)
```

## Description

`tpclose()` tears down the association between the caller and the resource manager to which it is linked. Since resource managers differ in their `close` semantics, the specific information needed to close a particular resource manager is placed in a configuration file.

If a resource manager is already closed (that is, `tpclose()` is called more than once), no action is taken and success is returned.

In a multithreaded application, a thread in the `TPINVALIDCONTEXT` state is not allowed to issue a call to `tpclose()`.

## Return Values

Upon failure, `tpclose()` returns -1 and sets `tperrno` to indicate the error condition.

## Errors

Upon failure, `tpclose()` fails and sets `tperrno` to one of the following values:

**[**`TPERMERR`**]**
: A resource manager failed to close correctly. More information concerning the reason a resource manager failed to close can be obtained by interrogating a resource manager in its own specific manner. Note that any calls to determine the exact nature of the error hinder portability.

**[**`TPEPROTO`**]**
: `tpclose()` was called in an improper context (for example, while the caller is in transaction mode).

**[**`TPESYSTEM`**]**
: A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**`TPEOS`**]**
: An operating system error has occurred.

## See Also

`tpopen(3c)`

# tpcommit(3c)

## Name

tpcommit()—Routine for committing current transaction.

## Synopsis

```
#include <atmi.h>
int tpcommit(long flags)
```

## Description

tpcommit() signifies the end of a transaction, using a two-phase commit protocol to coordinate participants. tpcommit() can be called only by the initiator of a transaction. If any of the participants cannot commit the transaction (for example, they call tpreturn() with TPFAIL), then the entire transaction is aborted and tpcommit() fails. That is, all of the work involved in the transaction is undone. If all participants agree to commit their portion of the transaction, then this decision is logged to stable storage and all participants are asked to commit their work.

Depending on the setting of the TP_COMMIT_CONTROL characteristic (see tpscmt(3c)), tpcommit() can return successfully either after the commit decision has been logged or after the two-phase commit protocol has completed. If tpcommit() returns after the commit decision has been logged but before the second phase has completed (TP_CMT_LOGGED), then all participants have agreed to commit the work they did on behalf of the transaction and should fulfill their promise to commit the transaction during the second phase. However, because tpcommit() is returning before the second phase has completed, there is a hazard that one or more of the participants can heuristically complete their portion of the transaction (in a manner that is not consistent with the commit decision) even though the function has returned success.

If the TP_COMMIT_CONTROL characteristic is set such that tpcommit() returns after the two-phase commit protocol has completed (TP_CMT_COMPLETE), then its return value reflects the exact status of the transaction (that is, whether the transaction heuristically completed or not).

Note that if only a single resource manager is involved in a transaction, then a one-phase commit is performed (that is, the resource manager is not asked whether or not it can commit; it is simply told to commit). In this case, the TP_COMMIT_CONTROL characteristic has no bearing and tpcommit() will return heuristic outcomes if present.

If tpcommit() is called while call descriptors exist for outstanding replies, then upon return from the function, the transaction is aborted and those descriptors associated with the caller's transaction are no longer valid. Call descriptors not associated with the caller's transaction remain valid.

tpcommit() must be called after all connections associated with the caller's transaction are closed (otherwise TPEABORT is returned, the transaction is aborted and these connections are disconnected in a disorderly fashion with a TPEV_DISCONIMM event). Connections opened before tpbegin() or with the TPNOTRAN flag (that is, connections not in transaction mode) are not affected by calls to tpcommit() or tpabort().

Currently, tpcommit()'s sole argument, *flags*, is reserved for future use and must be set to 0.

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tpcommit().

## Return Values

Upon failure, tpcommit() returns -1 and sets tperrno to indicate the error condition.

## Errors

Upon failure, tpcommit() sets tperrno to one of the following values:

**[**TPEABORT**]**
: The transaction could not commit because either the work performed by the initiator or by one or more of its participants could not commit. This error is also returned if tpcommit() is called with outstanding replies or open conversational connections.

**[**TPEHAZARD**]**
: Due to some failure, the work done on behalf of the transaction could have been heuristically completed.

**[**TPEHEURISTIC**]**
: Due to a heuristic decision, the work done on behalf of the transaction was partially committed and partially aborted.

**[**TPEINVAL**]**
: *flags* is not equal to 0. The caller's transaction is not affected.

**[**TPEOS**]**
: An operating system error has occurred.

**[**TPEPROTO**]**
: tpcommit() was called in an improper context (for example, by a participant).

**[**TPESYSTEM**]**
: A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPETIME**]**

The transaction has timed out and its status is unknown: it may have been either committed or aborted. If a transaction has timed out and its status is known to be aborted, then TPEABORT is returned.

### Notices

When using tpbegin(), tpcommit(), and tpabort() to delineate a BEA Tuxedo ATMI system transaction, it is important to remember that only the work done by a resource manager that meets the XA interface (and is linked to the caller appropriately) has transactional properties. All other operations performed in a transaction are not affected by either tpcommit() or tpabort(). See buildserver(1) for details on linking resource managers that meet the XA interface into a server such that operations performed by that resource manager are part of a BEA Tuxedo ATMI system transaction.

### See Also

tpabort(3c), tpbegin(3c), tpconnect(3c), tpgetlev(3c), tpreturn(3c), tpscmt(3c)

## tpconnect(3c)

### Name

tpconnect()—Routine for establishing a conversational service connection.

### Synopsis

```
#include <atmi.h>

int tpconnect(char *svc, char *data, long len, long flags)
```

### Description

tpconnect() allows a program to set up a half-duplex connection to a conversational service, *svc*. The name must be one of the conversational service names posted by a conversational server.

As part of setting up a connection, the caller can pass application-defined data to the listening program. If the caller chooses to pass data, then *data* must point to a buffer previously allocated by tpalloc(). *len* specifies how much of the buffer to send. Note that if *data* points to a buffer of a type that does not require a length to be specified, (for example, an FML fielded buffer), then *len* is ignored (and may be 0). Also, *data* can be NULL in which case *len* is ignored (no

application data is passed to the conversational service). The type and subtype of *data* must match one of the types and subtypes recognized by *svc*. *data* and *len* are passed to the conversational service via the TPSVCINFO structure with which the service is invoked; the service does not have to call tprecv() to get the data.

The following is a list of valid *flags*:

TPNOTRAN
: If the caller is in transaction mode and this flag is set, then when *svc* is invoked, it is not performed on behalf of the caller's transaction. Note that *svc* may still be invoked in transaction mode but it will not be the same transaction: a *svc* may have as a configuration attribute that it is automatically invoked in transaction mode. A caller in transaction mode that sets this flag is still subject to the transaction timeout (and no other). If a service fails that was invoked with this flag, the caller's transaction is not affected.

TPSENDONLY
: The caller wants the connection to be set up initially such that it can only send data and the called service can only receive data (that is, the caller initially has control of the connection). Either TPSENDONLY or TPRECVONLY must be specified.

TPRECVONLY
: The caller wants the connection to be set up initially such that it can only receive data and the called service can only send data (that is, the service being called initially has control of the connection). Either TPSENDONLY or TPRECVONLY must be specified.

TPNOBLOCK
: The connection is not established and the data is not sent if a blocking condition exists (for example, the data buffers through which the message is sent are full). Note that this flag applies only to the send portion of tpconnect(); the function may block waiting for an acknowledgement from the server. When TPNOBLOCK is not specified and a blocking condition exists, the caller blocks until the condition subsides or a blocking timeout or transaction timeout occurs.

TPNOTIME
: This flag signifies that the caller is willing to block indefinitely and wants to be immune to blocking timeouts. Transaction timeouts will still affect the program.

TPSIGRSTRT
: If a signal interrupts any underlying system calls, then the interrupted call is reissued.

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tpconnect().

## Return Values

Upon successful completion, `tpconnect()` returns a descriptor that is used to refer to the connection in subsequent calls. Otherwise it returns -1 and sets `tperrno` to indicate the error condition.

## Errors

Upon failure, `tpconnect()` sets `tperrno` to one of the following values. (Unless otherwise noted, failure does not affect the caller's transaction, if one exists.)

**[**`TPEBLOCK`**]**
A blocking condition exists and `TPNOBLOCK` was specified.

**[**`TPEINVAL`**]**
Invalid arguments were given (for example, *`svc`* is NULL, *`data`* is non-NULL and does not point to a buffer allocated by `tpalloc()`, `TPSENDONLY` or `TPRECVONLY` was not specified in *`flags`*, or *`flags`* are otherwise invalid).

**[**`TPEITYPE`**]**
The type and subtype of *`data`* is not one of the allowed types and subtypes that *`svc`* accepts.

**[**`TPELIMIT`**]**
The caller's request was not sent because the maximum number of outstanding connections has been reached.

**[**`TPENOENT`**]**
Cannot initiate a connection to *`svc`* because it does not exist or is not a conversational service.

**[**`TPEOS`**]**
An operating system error has occurred.

**[**`TPEPROTO`**]**
`tpconnect()` was called improperly.

**[**`TPESYSTEM`**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**`TPETIME`**]**
This error code indicates that either a timeout has occurred or `tpconnect()` has been attempted, in spite of the fact that the current transaction is already marked rollback only.

If the caller is in transaction mode, then either the transaction is already rollback only or a transaction timeout has occurred. The transaction is marked abort-only. If the caller is not in transaction mode, a blocking timeout has occurred. (A blocking timeout cannot occur if TPNOBLOCK and/or TPNOTIME is specified.)

If a transaction timeout has occurred, then, with one exception, any attempts to start new conversations, send new requests, or receive outstanding replies will fail with TPETIME until the transaction has been aborted. The exception is a request that does not block, expects no reply, and is not sent on behalf of the caller's transaction (that is, tpacall() with TPNOTRAN, TPNOBLOCK, and TPNOREPLY set).

When a service fails inside a transaction, the transaction is put into the TX_ROLLBACK_ONLY state. This state is treated, for most purposes, as though it were equivalent to a timeout. All further ATMI calls for this transaction (with the exception of those issued in the circumstances described in the previous paragraph) will fail with TPETIME.

**[**TPETRAN**]**
*svc* belongs to a program that does not support transactions and TPNOTRAN was not set.

**[**TPGOTSIG**]**
A signal was received and TPSIGRSTRT was not specified.

### See Also

tpalloc(3c),tpdiscon(3c),tprecv(3c),tpsend(3c),tpservice(3c)

## tpconvert(3c)

### Name

tpconvert()—Converts structures to/from string representations.

### Synopsis

```
#include <atmi.h>
#include <xa.h>

int tpconvert(char *strrep, char *binrep, long flags)
```

### Description

tpconvert() converts the string representation of interface structures (*strrep*) to or from the binary representation (*binrep*).

Both the direction of the conversion and the interface structure type are determined from the *flags* argument. To convert a structure from binary representation to string representation, the programmer must set the TPTOSTRING bit in *flags*. To convert a structure from string to binary the programmer must clear the bit. The following flags are defined to indicate the particular structure type to be converted; only one may be specified at a time:

TPCONVCLTID
: Convert CLIENTID (see atmi.h).

TPCONVTRANID
: Convert TPTRANID (see atmi.h).

TPCONVXID
: Convert XID (see xa.h).

For conversions from binary to string representation, *strrep* should be at least TPCONVMAXSTR characters in length.

Note that unequal string versions of TPTRANID and XID values may be considered *equal* by the system when accessing TM_MIB(5) classes that allow these values as key fields (for example, T_TRANSACTION or T_ULOG). Therefore, string values for these data types should not be fabricated or manipulated by application programs. TM_MIB(5) guarantees that only objects matching the global transaction identified by the string are returned when one of these values is used as a key field.

A thread in a multithreaded application may issue a call to tpconvert() while running in any context state, including TPINVALIDCONTEXT.

## Return Values

Upon failure, tpconvert() returns -1 and sets tperrno to indicate the error condition.

## Errors

Under the following conditions, tpconvert() fails and sets tperrno to one of the following values:

**[**TPEINVAL**]**
: Invalid arguments were specified. *strrep* or *binrep* is a NULL pointer, or *flags* does not indicate exactly one structure type.

**[**TPEOS**]**
: An operating system error has occurred. A numeric value representing the system call that failed is available in Uunixerr.

**[**TPESYSTEM**]**

A BEA Tuxedo system error has occurred. The exact nature of the error is written to userlog(3c).

### Portability

This interface is available only on BEA Tuxedo ATMI release 5.0 or later. This interface is available on workstation platforms.

### See Also

tpresume(3c), tpservice(3c), tpsuspend(3c), tx_info(3c), TM_MIB(5)

## tpconvmb(3c)

### Name

tpconvmb()—Converts encoding of characters in an input buffer to a named target encoding.

### Synopsis

```
#include <atmi.h>
extern int tperrno;
int
tpconvmb (char **bufp, int *len, char *target_encoding, long flags)
```

### Description

This function is used to convert an input buffer to a desired codeset encoding.

This function is added for user convenience and is not required for normal codeset data conversion that is done automatically.

The *bufp* argument is a valid pointer to an MBSTRING typed buffer message. This pointer will be reallocated internally if the size of the buffer is insufficient to handle the output data of the converted buffer.

The *len* argument, on input, contains the number of bytes that need to be converted. Upon successful completion of conversion it will contain the number of bytes used in bufp.

The *target_encoding* argument is the target codeset encoding name used to convert the typed buffer provided in the bufp message.

The *flags* argument is not used by the Tuxedo conversion code. It will be passed along to the buffer type switch function for user defined conversion functions.

### Return Values

Upon success, tpconvmb() returns 0. This function returns -1 on error and sets tperrno as described below. The function may fail for the following reasons.

**[**TPEINVAL**]**
*target_encoding*, *len*, or *bufp* arguments are NULL. *len* or *target_encoding* is invalid.

**[**TPEPROTO**]**
*bufp* translates to a Tuxedo buffer that does not have a buffer typeswitch conversion function

**[**TPESYSTEM**]**
A Tuxedo system error has occurred. (e.g. *bufp* does not correspond to a valid Tuxedo buffer).

**[**TPEOS**]**
An operating system error has occurred. A numeric value representing the system call that failed is available in Uunixerr.

### See Also

tpalloc(3c), tpgetmbenc(3c), tpsetmbenc(3c)

## tpcryptpw(3c)

### Name

tpcryptpw()—Encrypts the application password in an administrative request.

### Synopsis

```
#include <atmi.h>
#include <fml32.h>

int tpcryptpw(FBFR32 *buf)
```

### Description

tpcryptpw() is used to encrypt the application password stored in an administrative request buffer prior to sending the request for servicing. Application passwords are stored as string values using the FML32 field identifier TA_PASSWORD. This encryption is necessary to insure that clear

text passwords are not compromised and that appropriate propagation of the update can take place to all active application sites. Additional system fields may be added to the callers buffer and existing fields may be modified to satisfy the request.

A thread in a multithreaded application may issue a call to `tpcryptpw()` while running in any context state, including `TPINVALIDCONTEXT`.

## Return Values

Upon failure, `tpcryptpw()` returns -1 and sets `tperrno` to indicate the error condition.

## Errors

Upon failure, `tpcryptpw()` sets `tperrno` to one of the following values:

**[**`TPEINVAL`**]**
Invalid arguments were specified. The *`buf`* value is NULL, does not point to a FML32 typed buffer or appdir could not be determined from the input buffer or the environment.

**[**`TPEPERM`**]**
The calling process did not have the appropriate permissions necessary to perform the requested task.

**[**`TPEOS`**]**
An operating system error has occurred. A numeric value representing the system call that failed is available in `Uunixerr`.

**[**`TPESYSTEM`**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to `userlog(3c)`.

## Portability

This interface is available only on UNIX system sites running BEA Tuxedo ATMI release 5.0 or later. This interface is not available to Workstation clients.

## Files

`${TUXDIR}/lib/libtmib.a`, `${TUXDIR}/lib/libtmib.so.`*`rel`*

## See Also

`MIB(5)`, `TM_MIB(5)`

*Setting Up a BEA Tuxedo Application*

*Administering a BEA Tuxedo Application at Run Time*

# tpdequeue(3c)

## Name

tpdequeue()—Routine to dequeue a message from a queue.

## Synopsis

```
#include <atmi.h>
int tpdequeue(char *qspace, char *qname, TPQCTL *ctl, char **data, long
*len, long flags)
```

## Description

tpdequeue() takes a message for processing from the queue named by *qname* in the *qspace* queue space.

By default, the message at the top of the queue is dequeued. The order of messages on the queue is defined when the queue is created. The application can request a particular message for dequeuing by specifying its message identifier or correlation identifier using the *ctl* parameter. *ctl* flags can also be used to indicate that the application wants to wait for a message, in the case when a message is not currently available. It is possible to use the *ctl* parameter to look at a message without removing it from the queue or changing its relative position on the queue. See the section below describing this parameter.

*data* is the address of a pointer to the buffer into which a message is read, and *len* points to the length of that message. **data* must point to a buffer originally allocated by tpalloc(). If a message is larger than the buffer passed to tpdequeue, the buffer is increased in size to accommodate the message. To determine whether a message buffer changed in size, compare its (total) size before tpdequeue() was issued with **len*. If **len* is larger, then the buffer has grown; otherwise, the buffer has not changed size. Note that **data* may change for reasons other than the buffer's size increased. If **len* is 0 upon return, then the message dequeued has no data portion and neither **data* nor the buffer it points to were modified. It is an error for **data* or *len* to be NULL.

The message is dequeued in transaction mode if the caller is in transaction mode and the TPNOTRAN flag is not set. This has the effect that if tpdequeue() returns successfully and the caller's transaction is committed successfully, then the message is removed from the queue. If the caller's transaction is rolled back either explicitly or as the result of a transaction timeout or some communication error, then the message will be left on the queue (that is, the removal of the message from the queue is also rolled back). It is not possible to enqueue and dequeue the same message within the same transaction.

The message is not dequeued in transaction mode if either the caller is not in transaction mode, or the TPNOTRAN flag is set. When not in transaction mode, if a communication error or a timeout occurs, the application will not know whether or not the message was successfully dequeued and the message may be lost.

The following is a list of valid *flags*:

TPNOTRAN
: If the caller is in transaction mode and this flag is set, the message is not dequeued within the caller's transaction. A caller in transaction mode that sets this flag is still subject to the transaction timeout (and no other) when dequeuing the message. If message dequeuing fails, the caller's transaction is not affected.

TPNOBLOCK
: The message is not dequeued if a blocking condition exists. If this flag is set and a blocking condition exists such as the internal buffers into which the message is transferred are full, the call fails and tperrno is set to TPEBLOCK. If this flag is set and a blocking condition exists because the target queue is opened *exclusively* by another application, the call fails, tperrno is set to TPEDIAGNOSTIC, and the diagnostic field of the TPQCTL structure is set to QMESHARE. In the latter case, the other application, which is based on a BEA product other than the BEA Tuxedo ATMI system, opened the queue for exclusive read and/or write using the Queuing Services API (QSAPI).

  When TPNOBLOCK is not set and a blocking condition exists, the caller blocks until the condition subsides or a timeout occurs (either transaction or blocking timeout). This blocking condition does not include blocking on the queue itself if the TPQWAIT option in *flags* (of the TPQCTL structure) is specified.

TPNOTIME
: Setting this flag signifies that the caller is willing to block indefinitely and wants to be immune to blocking timeouts. Transaction timeouts may still occur.

TPNOCHANGE
: When this flag is set, the type of the buffer pointed to by **data* is not allowed to change. By default, if a buffer is received that differs in type from the buffer pointed to by **data*, then **data*'s buffer type changes to the received buffer's type so long as the receiver recognizes the incoming buffer type. That is, the type and subtype of the dequeued message must match the type and subtype of the buffer pointed to by **data*.

TPSIGRSTRT
: Setting this flag indicates that any underlying system calls that are interrupted by a signal should be reissued. When this flag is not set and a signal interrupts a system call, the call fails and sets tperrno to TPGOTSIG.

If tpdequeue() returns successfully, the application can retrieve additional information about the message using the *ctl* data structure. The information may include the message identifier for the dequeued message; a correlation identifier that should accompany any reply or failure message so that the originator can correlate the message with the original request; the quality of service the message was delivered with, the quality of service any replies to the message should be delivered with; the name of a reply queue if a reply is desired; and the name of the failure queue on which the application can queue information regarding failure to dequeue the message. These are described below.

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tpdequeue().

## Control Parameter

The TPQCTL structure is used by the application program to pass and retrieve parameters associated with dequeuing the message. The *flags* element of TPQCTL is used to indicate what other elements in the structure are valid.

On input to tpdequeue(), the following elements may be set in the TPQCTL structure:

```
long flags;             /* indicates which of the values
                         * are set */
char msgid[32];         /* ID of message to dequeue */
char corrid[32];        /* correlation identifier of
                         * message to dequeue */
```

The following is a list of valid bits for the *flags* parameter controlling input information for tpdequeue():

TPNOFLAGS
: No flags are set. No information is taken from the control structure.

TPQGETBYMSGID
: Setting this flag requests that the message with the message identifier specified by *ctl->msgid* be dequeued. The message identifier may be acquired by a prior call to tpenqueue(3c). Note that a message identifier changes if the message has moved from one queue to another. Note also that the entire 32 bytes of the message identifier value are significant, so the value specified by *ctl->msgid* must be completely initialized (for example, padded with NULL characters).

TPQGETBYCORRID
: Setting this flag requests that the message with the correlation identifier specified by *ctl->corrid* be dequeued. The correlation identifier is specified by the application when enqueuing the message with tpenqueue(). Note that the entire 32 bytes of the

correlation identifier value are significant, so the value specified by *ctl->corrid* must be completely initialized (for example, padded with NULL characters).

TPQWAIT

Setting this flag indicates that an error should not be returned if the queue is empty. Instead, the process should wait until a message is available. If TPQWAIT is set in conjunction with TPQGETBYMSGID or TPQGETBYCORRID, it indicates that an error should not be returned if no message with the specified message identifier or correlation identifier is present in the queue. Instead, the process should wait until a message meeting the criteria is available. The process is still subject to the caller's transaction timeout, or, when not in transaction mode, the process is subject to the timeout specified on the TMQUEUE process by the -t option.

If a message matching the desired criteria is not immediately available and the configured action resources are exhausted, tpdequeue returns -1, tperrno is set to TPEDIAGNOSTIC, and the diagnostic field of the TPQCTL structure is set to QMESYSTEM.

Note that each tpdequeue() request specifying the TPQWAIT control parameter requires that a queue manager (TMQUEUE) action object be available if a message satisfying the condition is not immediately available. If an action object is not available, the tpdequeue() request fails. The number of available queue manager actions are specified when a queue space is created or modified. When a waiting dequeue request completes, the associated action object associated is made available for another request.

TPQPEEK

If this flag is set, the specified message is read but is not removed from the queue. This flag implies the TPNOTRAN flag has been set for the tpdequeue() operation. That is, non-destructive dequeuing is non-transactional. Note that it is not possible to read messages enqueued or dequeued within a transaction before the transaction completes.

When a thread is non-destructively dequeuing a message using TPQPEEK, the message may not be seen by other non-blocking dequeuers for the brief time the system is processing the non-destructive dequeue request. This includes dequeuers using specific selection criteria (such as message identifier and correlation identifier) that are looking for the message currently being non-destructively dequeued.

On output from tpdequeue(), the following elements may be set in the TPQCTL structure:

```
long flags;             /* indicates which of the values
                         * should be set */

long priority;          /* enqueue priority */
char msgid[32];         /* ID of message dequeued */
```

```
char corrid[32];        /* correlation identifier used to
                         * identify the message */
long delivery_qos;      /* delivery quality of service */
long reply_qos;         /* reply message quality of service */
char replyqueue[16];    /* queue name for reply */
char failurequeue[16]; /* queue name for failure */
long diagnostic;        /* reason for failure */
long appkey;            /* application authentication client
                         * key */
long urcode;            /* user-return code */
CLIENTID cltid;         /* client identifier for originating
                         * client */
```

The following is a list of valid bits for the *flags* parameter controlling output information from tpdequeue(). For any of these bits, if the flag bit is turned on when tpdequeue() is called, the associated element in the structure is populated with the value provided when the message was queued, and the bit remains set. If a value is not available or the bit is not set when tpdequeue() is called, tpdequeue() completes with the flag turned off.

TPQPRIORITY
: If this flag is set, the call to tpdequeue() is successful, and the message was queued with an explicit priority, then the priority is stored in *ctl→priority*. The priority is in the range 1 to 100, inclusive, and the higher the number, the higher the priority (that is, a message with a higher number is dequeued before a message with a lower number). For queues not ordered by priority, the value is informational.

  If no priority was explicitly specified when the message was queued and the call to tpdequeue() is successful, the priority for the message is 50.

TPQMSGID
: If this flag is set and the call to tpdequeue() is successful, the message identifier is stored in *ctl→msgid*. The entire 32 bytes of the message identifier value are significant.

TPQCORRID
: If this flag is set, the call to tpdequeue() is successful, and the message was queued with a correlation identifier, then the correlation identifier is stored in *ctl→corrid*. The entire 32 bytes of the correlation identifier value are significant. Any BEA Tuxedo ATMI /Q provided reply to a message has the correlation identifier of the original request message.

TPQDELIVERYQOS

If this flag is set, the call to tpdequeue() is successful, and the message was queued with a delivery quality of service, then the flag—TPQQOSDEFAULTPERSIST, TPQQOSPERSISTENT, or TPQQOSNONPERSISTENT—is stored in *ctl->delivery_qos*. If no delivery quality of service was explicitly specified when the message was queued, the default delivery policy of the target queue dictates the delivery quality of service for the message.

TPQREPLYQOS

If this flag is set, the call to tpdequeue() is successful, and the message was queued with a reply quality of service, then the flag—TPQQOSDEFAULTPERSIST, TPQQOSPERSISTENT, or TPQQOSNONPERSISTENT—is stored in *ctl->reply_qos*. If no reply quality of service was explicitly specified when the message was queued, the default delivery policy of the *ctl->replyqueue* queue dictates the delivery quality of service for any reply.

Note that the default delivery policy is determined when the reply to a message is enqueued. That is, if the default delivery policy of the reply queue is modified between the time that the original message is enqueued and the reply to the message is enqueued, the policy used is the one in effect when the reply is finally enqueued.

TPQREPLYQ

If this flag is set, the call to tpdequeue() is successful, and the message was queued with a reply queue, then the name of the reply queue is stored in *ctl≫replyqueue*. Any reply to the message should go to the named reply queue within the same queue space as the request message.

TPQFAILUREQ

If this flag is set, the call to tpdequeue() is successful, and the message was queued with a failure queue, then the name of the failure queue is stored in *ctl≫failurequeue*. Any failure message should go to the named failure queue within the same queue space as the request message.

The following remaining bits for the *flags* parameter are cleared (set to zero) when tpdequeue() is called: TPQTOP, TPQBEFOREMSGID, TPQTIME_ABS, TPQTIME_REL, TPQEXPTIME_ABS, TPQEXPTIME_REL, and TPQEXPTIME_NONE. These bits are valid bits for the *flags* parameter controlling input information for tpenqueue().

If the call to tpdequeue() failed and tperrno is set to TPEDIAGNOSTIC, a value indicating the reason for failure is returned in *ctl≫diagnostic*. The possible values are defined below in the Diagnostics section.

Additionally on output, if the call to tpdequeue() is successful, *ctl≫appkey* is set to the application authentication key, *ctl≫cltid* is set to the identifier for the client originating the

request, and *ctl->urcode* is set to the user-return code value that was set when the message was enqueued.

If the *ctl* parameter is NULL, the input flags are considered to be TPNOFLAGS, and no output information is made available to the application program.

## Return Values

Upon failure, tpdequeue() returns -1 and sets tperrno to indicate the error condition.

## Errors

Upon failure, tpdequeue() sets tperrno to one of the following values. (Unless otherwise noted, failure does not affect the caller's transaction, if one exists.)

**[**TPEINVAL**]**

Invalid arguments were given (for example, *qname* is NULL, *data* does not point to space allocated with tpalloc() or *flags* are invalid).

**[**TPENOENT**]**

Cannot access the *qspace* because it is not available (that is, the associated TMQUEUE(5) server is not available), or cannot start a global transaction due to the lack of entries in the Global Transaction Table (GTT).

**[**TPEOTYPE**]**

Either the type and subtype of the dequeued message are not known to the caller; or, TPNOCHANGE was set in *flags* and the type and subtype of **data* do not match the type and subtype of the dequeued message. In either case, **data*, its contents, and **len* are *not* changed. When the call is made in transaction mode and this error occurs, the transaction is marked abort-only, and the message remains on the queue.

**[**TPETIME**]**

This error code indicates that either a timeout has occurred or tpdequeue() has been attempted, in spite of the fact that the current transaction is already marked rollback only.

If the caller is in transaction mode, then either the transaction is already rollback only or a transaction timeout has occurred. The transaction is marked abort-only. If the caller is not in transaction mode, a blocking timeout has occurred. (A blocking timeout cannot occur if TPNOBLOCK and/or TPNOTIME is specified.) In either case, no changes are made to *data, its contents, or *len.

If a transaction timeout has occurred, then, with one exception, any attempts to perform further conversational work, send new requests, or receive outstanding replies will fail with TPETIME until the transaction has been aborted. The exception is a request that does

not block, expects no reply, and is not sent on behalf of the caller's transaction (that is, `tpacall()` with `TPNOTRAN`, `TPNOBLOCK`, and `TPNOREPLY` set).

When a service fails inside a transaction, the transaction is put into the `TX_ROLLBACK_ONLY` state. This state is treated, for most purposes, as though it were equivalent to a timeout. All further ATMI calls for this transaction (with the exception of those issued in the circumstances described in the previous paragraph) will fail with `TPETIME`.

**[**`TPEBLOCK`**]**
A blocking condition exists and `TPNOBLOCK` was specified.

**[**`TPGOTSIG`**]**
A signal was received and `TPSIGRSTRT` was not specified.

**[**`TPEPROTO`**]**
`tpdequeue()` was called improperly. There is no effect on the queue or the transaction.

**[**`TPESYSTEM`**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file. There is no effect on the queue.

**[**`TPEOS`**]**
An operating system error has occurred. There is no effect on the queue.

**[**`TPEDIAGNOSTIC`**]**
Dequeuing a message from the specified queue failed. The reason for failure can be determined by the diagnostic value returned via *`ctl`* structure.

## Diagnostic

The following diagnostic values are returned during the dequeuing of a message:

**[**`QMEINVAL`**]**
An invalid flag value was specified.

**[**`QMEBADRMID`**]**
An invalid resource manager identifier was specified.

**[**`QMENOTOPEN`**]**
The resource manager is not currently open.

**[**QMETRAN**]**
The call was not in transaction mode or was made with the TPNOTRAN flag set and an error occurred trying to start a transaction in which to dequeue the message. This diagnostic is not returned by queue managers from BEA Tuxedo release 7.1 or later.

**[**QMEBADMSGID**]**
An invalid message identifier was specified for dequeuing.

**[**QMESYSTEM**]**
A system error has occurred. The exact nature of the error is written to a log file.

**[**QMEOS**]**
An operating system error has occurred.

**[**QMEABORTED**]**
The operation was aborted. When executed within a global transaction, the global transaction has been marked rollback-only. Otherwise, the queue manager aborted the operation.

**[**QMEPROTO**]**
A dequeue was done when the transaction state was not active.

**[**QMEBADQUEUE**]**
An invalid or deleted queue name was specified.

**[**QMENOMSG**]**
No message was available for dequeuing. Note that it is possible that the message exists on the queue and another application process has read the message from the queue. In this case, the message may be put back on the queue if that other process rolls back the transaction.

**[**QMEINUSE**]**
When dequeuing a message by message identifier or correlation identifier, the specified message is in use by another transaction. Otherwise, all messages currently on the queue are in use by other transactions. This diagnostic is not returned by queue managers from BEA Tuxedo release 7.1 or later.

**[**QMESHARE**]**
When dequeuing a message from a specified queue, the specified queue is opened *exclusively* by another application. The other application is one based on a BEA product other than the BEA Tuxedo system that opened the queue for exclusive read and/or write using the Queuing Services API (QSAPI).

### See Also

qmadmin(1), tpalloc(3c),tpenqueue(3c),APPQ_MIB(5), TMQUEUE(5)

## tpdiscon(3c)

### Name

tpdiscon()—Routine for taking down a conversational service connection.

### Synopsis

```
#include <atmi.h>
int tpdiscon(int cd)
```

### Description

tpdiscon() immediately tears down the connection specified by *cd* and generates a TPEV_DISCONIMM event on the other end of the connection.

tpdiscon() can be called only by the initiator of the conversation. tpdiscon() cannot be called within a conversational service on the descriptor with which it was invoked. Rather, a conversational service must use tpreturn() to signify that it has completed its part of the conversation. Similarly, even though a program communicating with a conversational service can issue tpdiscon(), the preferred way is to let the service tear down the connection in tpreturn(); doing so ensures correct results.

tpdiscon() causes the connection to be torn down immediately (that is, abortive rather than orderly). Any data that has not yet reached its destination may be lost. tpdiscon() can be issued even when the program on the other end of the connection is participating in the caller's transaction. In this case, the transaction must be aborted. Also, the caller does not need to have control of the connection when tpdiscon() is called.

### Return Values

Upon failure, tpdiscon() returns -1 and sets tperrno to indicate the error condition.

### Errors

Upon failure, tpdiscon() sets tperrno to one of the following values:

**[**TPEBADDESC**]**
*cd* is invalid or is the descriptor with which a conversational service was invoked.

**[**TPETIME**]**

This error code indicates that either a timeout has occurred or tpdiscon() has been attempted, in spite of the fact that the current transaction is already marked rollback only.

If the caller is in transaction mode, then either the transaction is already rollback only or a transaction timeout has occurred. The transaction is marked abort-only. (Note that calling tpdiscon() on a connection in the caller's transaction would have resulted in the transaction being marked abort-only, even if tpdiscon() had succeeded.)

If the caller is not in transaction mode, a blocking timeout has occurred.

If a transaction timeout has occurred, then, with one exception, any attempts to perform further conversational work, send new requests, or receive outstanding replies will fail with TPETIME until the transaction has been aborted. The exception is a request that does not block, expects no reply, and is not sent on behalf of the caller's transaction (that is, tpacall() with TPNOTRAN, TPNOBLOCK, and TPNOREPLY set).

When a service fails inside a transaction, the transaction is put into the TX_ROLLBACK_ONLY state. This state is treated, for most purposes, as though it were equivalent to a timeout. All further ATMI calls for this transaction (with the exception of those issued in the circumstances described in the previous paragraph) will fail with TPETIME.

**[**TPEPROTO**]**

tpdiscon() was called improperly.

**[**TPESYSTEM**]**

A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file. The descriptor is no longer valid.

**[**TPEOS**]**

An operating system error has occurred. The descriptor is no longer valid.

### See Also

tpabort(3c), tpcommit(3c), tpconnect(3c), tprecv(3c), tpreturn(3c), tpsend(3c)

## tpenqueue(3c)

### Name

tpenqueue()—Routine to enqueue a message.

## Synopsis

```
#include <atmi.h>
int tpenqueue(char *qspace, char *qname, TPQCTL *ctl, char *data, long len,
long flags)
```

## Description

`tpenqueue()` stores a message on the queue named by *`qname`* in the *`qspace`* queue space. A queue space is a collection of queues, one of which must be *`qname`*.

When the message is intended for a BEA Tuxedo ATMI system server, the *`qname`* matches the name of a service provided by the server. The system provided server, `TMQFORWARD(5)`, provides a default mechanism for dequeuing messages from the queue and forwarding them to servers that provide a service matching the queue name. If the originator expects a reply, then the reply to the forwarded service request is stored on the originator's queue, unless otherwise specified. The originator will dequeue the reply message at a subsequent time. Queues can also be used for a reliable message transfer mechanism between any pair of BEA Tuxedo ATMI system processes (clients and/or servers). In this case, the queue name does not match a service name but some agreed upon name for transferring the message.

If *`data`* is non-NULL, it must point to a buffer previously allocated by `tpalloc()` and *`len`* should specify the amount of data in the buffer that should be queued. Note that if *`data`* points to a buffer of a type that does not require a length to be specified (for example, an `FML` fielded buffer), then *`len`* is ignored. If *`data`* is NULL, *`len`* is ignored and a message is queued with no data portion.

The message is queued at the priority defined for *`qspace`* unless overridden by a previous call to `tpsprio()`.

If the caller is within a transaction and the `TPNOTRAN` flag is not set, the message is queued in transaction mode. This has the effect that if `tpenqueue()` returns successfully and the caller's transaction is committed successfully, then the message is guaranteed to be available subsequent to the transaction completing. If the caller's transaction is rolled back either explicitly or as the result of a transaction timeout or some communication error, then the message will be removed from the queue (that is, the placing of the message on the queue is also rolled back). It is not possible to enqueue then dequeue the same message within the same transaction.

The message is not queued in transaction mode if either the caller is not in transaction mode, or the `TPNOTRAN` flag is set. Once `tpenqueue()` returns successfully, the submitted message is guaranteed to be in the queue. When not in transaction mode, if a communication error or a

timeout occurs, the application will not know whether or not the message was successfully stored on the queue.

The order in which messages are placed on the queue is controlled by the application via *ctl* data structure as described below; the default queue ordering is set when the queue is created.

The following is a list of valid *flags*:

TPNOTRAN
: If the caller is in transaction mode and this flag is set, the message is not queued within the caller's transaction. A caller in transaction mode that sets this flag is still subject to the transaction timeout (and no other) when queuing the message. If message queuing fails, the caller's transaction is not affected.

TPNOBLOCK
: The message is not enqueued if a blocking condition exists. If this flag is set and a blocking condition exists such as the internal buffers into which the message is transferred are full, the call fails and tperrno is set to TPEBLOCK. If this flag is set and a blocking condition exists because the target queue is opened *exclusively* by another application, the call fails, tperrno is set to TPEDIAGNOSTIC, and the diagnostic field of the TPQCTL structure is set to QMESHARE. In the latter case, the other application, which is based on a BEA product other than the BEA Tuxedo ATMI system, opened the queue for exclusive read and/or write using the Queuing Services API (QSAPI).

  When TPNOBLOCK is not set and a blocking condition exists, the caller blocks until the condition subsides or a timeout occurs (either transaction or blocking timeout). If a timeout occurs, the call fails and tperrno is set to TPETIME.

TPNOTIME
: Setting this flag signifies that the caller is willing to block indefinitely and wants to be immune to blocking timeouts. Transaction timeouts may still occur.

TPSIGRSTRT
: If this flag is set and a signal interrupts any underlying system calls, the interrupted system call is reissued. If TPSIGRSTRT is not set and a signal interrupts a system call, tpenqueue() fails and tperrno is set to TPGOTSIG.

Additional information about queuing the message can be specified via *ctl* data structure. This information includes values to override the default queue ordering placing the message at the top of the queue or before an enqueued message; an absolute or relative time after which a queued message is made available; an absolute or relative time when a message expires and is removed from the queue; the quality of service for delivering the message; the quality of service that any replies to the message should use; a correlation identifier that aids in correlating a reply or failure

message with the queued message; the name of a queue to which a reply should be enqueued; and the name of a queue to which any failure message should be enqueued.

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tpenqueue().

## Control Parameter

The TPQCTL structure is used by the application program to pass and retrieve parameters associated with enqueuing the message. The *flags* element of TPQCTL is used to indicate what other elements in the structure are valid.

On input to tpenqueue(), the following elements may be set in the TPQCTL structure:

```
long flags;             /* indicates which of the values
                         * are set */
long deq_time;          /* absolute/relative for dequeuing */
long priority;          /* enqueue priority */
long exp_time           /* expiration time */
long delivery_qos       /* delivery quality of service */
long reply_qos          /* reply quality of service */
long urcode;            /* user-return code */
char msgid[32];         /* ID of message before which to queue
                         * request */
char corrid[32];        /* correlation identifier used to
                         * identify the msg */
char replyqueue[16];    /* queue name for reply message */
char failurequeue[16]; /* queue name for failure message */
```

The following is a list of valid bits for the *flags* parameter controlling input information for tpenqueue():

TPNOFLAGS
: No flags or values are set. No information is taken from the control structure.

TPQTOP
: Setting this flag indicates that the queue ordering be overridden and the message placed at the top of the queue. This request may not be granted depending on whether or not the queue was configured to allow overriding the queue ordering. TPQTOP and TPQBEFOREMSGID are mutually exclusive flags.

TPQBEFOREMSGID

Setting this flag indicates that the queue ordering be overridden and the message placed in the queue before the message identified by *ctl->msgid*. This request may not be granted depending on whether or not the queue was configured to allow overriding the queue ordering. TPQTOP and TPQBEFOREMSGID are mutually exclusive flags. Note that the entire 32 bytes of the message identifier value are significant, so the value identified by *ctl->msgid* must be completely initialized (for example, padded with NULL characters).

TPQTIME_ABS

If this flag is set, the message is made available after the time specified by *ctl->deq_time*. The *deq_time* is an absolute time value as generated by time(2), mktime(3C), or gp_mktime(3c) (the number of seconds since 00:00:00 Universal Coordinated Time—UTC, January 1, 1970). TPQTIME_ABS and TPQTIME_REL are mutually exclusive flags. The absolute time is determined by the clock on the machine where the queue manager process resides.

TPQTIME_REL

If this flag is set, the message is made available after a time relative to the completion of the enqueuing operation. *ctl->deq_time* specifies the number of seconds to delay after the enqueuing completes before the submitted message should be available. TPQTIME_ABS and TPQTIME_REL are mutually exclusive flags.

TPQPRIORITY

If this flag is set, the priority at which the message should be enqueued is stored in *ctl->priority*. The priority must be in the range 1 to 100, inclusive. The higher the number, the higher the priority (that is, a message with a higher number is dequeued before a message with a lower number). For queues not ordered by priority, this value is informational.

If this flag is not set, the priority for the message is 50 by default.

TPQCORRID

If this flag is set, the correlation identifier value specified in *ctl->corrid* is available when a message is dequeued with tpdequeue(). This identifier accompanies any reply or failure message that is queued so that an application can correlate a reply with a particular request. Note that the entire 32 bytes of the correlation identifier value are significant, so the value specified in *ctl->corrid* must be completely initialized (for example, padded with NULL characters).

TPQREPLYQ

If this flag is set, a reply queue named in *ctl->replyqueue* is associated with the queued message. Any reply to the message will be queued to the named queue within the same

queue space as the request message. This string must be NULL terminated (maximum 15 characters in length).

TPQFAILUREQ

If this flag is set, a failure queue named in the *ctl->failurequeue* is associated with the queued message. If (1) the enqueued message is processed by TMQFORWARD(), (2) TMQFORWARD was started with the -d option, and (3) the service fails and returns a non-NULL reply, a failure message consisting of the reply and its associated tpurcode is enqueued to the named queue within the same queue space as the original request message. This string must be NULL-terminated (maximum 15 characters in length).

TPQDELIVERYQOS, TPQREPLYQOS

If the TPQDELIVERYQOS flag is set, the flags specified by *ctl->delivery_qos* control the quality of service for delivery of the message. In this case, one of three mutually exclusive flags— TPQQOSDEFAULTPERSIST, TPQQOSPERSISTENT, or TPQQOSNONPERSISTENT—must be set in *ctl->delivery_qos*. If TPQDELIVERYQOS is not set, the default delivery policy of the target queue dictates the delivery quality of service for the message.

If the TPQREPLYQOS flag is set, the flags specified by *ctl->reply_qos* control the quality of service for any reply to the message. In this case, one of three mutually exclusive flags—TPQQOSDEFAULTPERSIST, TPQQOSPERSISTENT, or TPQQOSNONPERSISTENT—must be set in *ctl->reply_qos*. The TPQREPLYQOS flag is used when a reply is returned from messages processed by TMQFORWARD. Applications not using TMQFORWARD to invoke services may use the TPQREPLYQOS flag as a hint for their own reply mechanism.

If TPQREPLYQOS is not set, the default delivery policy of the *ctl->replyqueue* queue dictates the delivery quality of service for any reply. Note that the default delivery policy is determined when the reply to a message is enqueued. That is, if the default delivery policy of the reply queue is modified between the time that the original message is enqueued and the reply to the message is enqueued, the policy used is the one in effect when the reply is finally enqueued.

The following is the list of valid flags for *ctl->delivery_qos* and *ctl->reply_qos*:

TPQQOSDEFAULTPERSIST

This flag specifies that the message is to be delivered using the default delivery policy specified on the target queue.

TPQQOSPERSISTENT

This flag specifies that the message is to be delivered in a persistent manner using the disk-based delivery method. Setting this flag overrides the default delivery policy specified on the target queue.

TPQQOSNONPERSISTENT

This flag specifies that the message is to be delivered in a non-persistent manner using the memory-based delivery method. Specifically, the message is queued in memory until it is dequeued. Setting this flag overrides the default delivery policy specified on the target queue. If the caller is transactional, non-persistent messages are enqueued within the caller's transaction, however, non-persistent messages are lost if the system is shut down, crashes, or the IPC shared memory for the queue space is removed.

TPQEXPTIME_ABS

If this flag is set, the message has an absolute expiration time, which is the absolute time when the message will be removed from the queue.
The absolute expiration time is determined by the clock on the machine where the queue manager process resides.

The absolute expiration time is indicated by the value stored in *ctl->exp_time*. The value of *ctl->exp_time* must be set to an absolute time value generated by time(2), mktime(3C), or gp_mktime(3c) (the number of seconds since 00:00:00 Universal Coordinated Time—UTC, January 1, 1970).

If an absolute time is specified that is earlier than the time of the enqueue operation, the operation succeeds, but the message is not counted for the purpose of calculating thresholds. If the expiration time is before the message availability time, the message is not available for dequeuing unless either the availability or expiration time is changed so that the availability time is before the expiration time. In addition, these messages are removed from the queue at expiration time even if they were never available for dequeuing. If a message expires while it is within a transaction, the expiration does not cause the transaction to fail. Messages that expire while being enqueued or dequeued within a transaction are removed from the queue when the transaction ends. There is no notification that the message has expired.

TPQEXPTIME_ABS, TPQEXPTIME_REL, and TPQEXPTIME_NONE are mutually exclusive flags. If none of these flags is set, the default expiration time associated with the target queue is applied to the message.

TPQEXPTIME_REL

If this flag is set, the message has a relative expiration time, which is the number of seconds *after* the message arrives at the queue that the message is removed from the queue. The relative expiration time is indicated by the value stored in *ctl->exp_time*.

If the expiration time is before the message availability time, the message is not available for dequeuing unless either the availability or expiration time is changed so that the availability time is before the expiration time. In addition, these messages are removed

from the queue at expiration time even if they were never available for dequeuing. The expiration of a message during a transaction, does not cause the transaction to fail. Messages that expire while being enqueued or dequeued within a transaction are removed from the queue when the transaction ends. There is no acknowledgment that the message has expired.

TPQEXPTIME_ABS, TPQEXPTIME_REL, and TPQEXPTIME_NONE are mutually exclusive flags. If none of these flags is set, the default expiration time associated with the target queue is applied to the message.

TPQEXPTIME_NONE

Setting this flag indicates that the message should not expire. This flag overrides any default expiration policy associated with the target queue. A message can be removed by dequeuing it or by deleting it via an administrative interface.

TPQEXPTIME_ABS, TPQEXPTIME_REL, and TPQEXPTIME_NONE are mutually exclusive flags. If none of these flags is set, the default expiration time associated with the target queue is applied to the message.

Additionally, the *urcode* element of TPQCTL can be set with a user-return code. This value will be returned to the application that dequeues the message.

On output from tpenqueue(), the following elements may be set in the TPQCTL structure:

```
long flags;         /* indicates which of the values
                                * are set */
char msgid[32];               /* ID of enqueued message */
long diagnostic;              /* indicates reason for failure */
```

The following is a valid bit for the *flags* parameter controlling output information from tpenqueue(). If this flag is turned on when tpenqueue() is called, the /Q server TMQUEUE(5) populates the associated element in the structure with a message identifier. If this flag is turned off when tpenqueue() is called, TMQUEUE() does *not* populate the associated element in the structure with a message identifier.

TPQMSGID

If this flag is set and the call to tpenqueue() is successful, the message identifier is stored in *ctl->msgid*. The entire 32 bytes of the message identifier value are significant, so the value stored in *ctl->msgid* is completely initialized (for example, padded with NULL characters). The actual padding character used for initialization varies between releases of the BEA Tuxedo ATMI /Q component.

The remaining members of the control structure are not used on input to tpenqueue().

If the call to tpenqueue() failed and tperrno is set to TPEDIAGNOSTIC, a value indicating the reason for failure is returned in *ctl->diagnostic*. The possible values are defined below in the Diagnostics section.

If this parameter is NULL, the input flags are considered to be TPNOFLAGS and no output information is made available to the application program.

## Return Values

Upon failure, tpenqueue() returns -1 and sets tperrno to indicate the error condition. Otherwise, the message has been successfully queued when tpenqueue() returns.

## Errors

Upon failure, tpenqueue() sets tperrno to one of the following values. (Unless otherwise noted, failure does not affect the caller's transaction, if one exists.)

**[**TPEINVAL**]**
: Invalid arguments were given (for example, *qspace* is NULL, *data* does not point to space allocated with tpalloc(), or *flags* are invalid).

**[**TPENOENT**]**
: Cannot access the *qspace* because it is not available (that is, the associated TMQUEUE(5) server is not available), or cannot start a global transaction due to the lack of entries in the Global Transaction Table (GTT).

**[**TPETIME**]**
: This error code indicates that either a timeout has occurred or tpenqueue() has been attempted, in spite of the fact that the current transaction is already marked rollback only.

  If the caller is in transaction mode, then either the transaction is already rollback only or a transaction timeout has occurred. The transaction is marked abort-only. If the caller is not in transaction mode, a blocking timeout has occurred. (A blocking timeout cannot occur if TPNOBLOCK and/or TPNOTIME is specified.)

  If a transaction timeout has occurred, then, with one exception, any attempts to send new requests or receive outstanding replies will fail with TPETIME until the transaction has been aborted. The exception is a request that does not block, expects no reply, and is not sent on behalf of the caller's transaction (that is, tpacall() with TPNOTRAN, TPNOBLOCK, and TPNOREPLY set).

  When a service fails inside a transaction, the transaction is put into the TX_ROLLBACK_ONLY state. This state is treated, for most purposes, as though it were equivalent to a timeout. All further ATMI calls for this transaction (with the exception of

those issued in the circumstances described in the previous paragraph) will fail with TPETIME.

**[**TPEBLOCK**]**
A blocking condition exists and TPNOBLOCK was specified.

**[**TPGOTSIG**]**
A signal was received and TPSIGRSTRT was not specified.

**[**TPEPROTO**]**
tpenqueue() was called improperly.

**[**TPESYSTEM**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPEOS**]**
An operating system error has occurred.

**[**TPEDIAGNOSTIC**]**
Enqueuing a message on the specified queue failed. The reason for failure can be determined by the diagnostic returned via *ctl*.

## Diagnostic

The following diagnostic values are returned during the enqueuing of a message:

**[**QMEINVAL**]**
An invalid flag value was specified.

**[**QMEBADRMID**]**
An invalid resource manager identifier was specified.

**[**QMENOTOPEN**]**
The resource manager is not currently open.

**[**QMETRAN**]**
The call was not in transaction mode or was made with the TPNOTRAN flag set and an error occurred trying to start a transaction in which to enqueue the message. This diagnostic is not returned by queue managers from BEA Tuxedo release 7.1 or later.

**[**QMEBADMSGID**]**
An invalid message identifier was specified.

**[**QMESYSTEM**]**
A system error occurred. The exact nature of the error is written to a log file.

**[**QMEOS**]**
An operating system error occurred.

**[**QMEABORTED**]**
The operation was aborted. When executed within a global transaction, the global transaction has been marked rollback-only. Otherwise, the queue manager aborted the operation.

**[**QMEPROTO**]**
An enqueue was done when the transaction state was not active.

**[**QMEBADQUEUE**]**
An invalid or deleted queue name was specified.

**[**QMENOSPACE**]**
Due to an insufficient resource, such as no space on the queue, the message with its required quality of service (persistent or non-persistent storage) was not enqueued. QMENOSPACE is returned when any of the following configured resources is exceeded: (1) the amount of disk (persistent) space allotted to the queue space, (2) the amount of memory (non-persistent) space allotted to the queue space, (3) the maximum number of simultaneously active transactions allowed for the queue space, (4) the maximum number of messages that the queue space can contain at any one time, (5) the maximum number of concurrent actions that the Queuing Services component can handle, or (6) the maximum number of authenticated users that may concurrently use the Queuing Services component.

**[**QMERELEASE**]**
An attempt was made to enqueue a message to a queue manager that is from a version of the BEA Tuxedo system that does not support a newer feature.

**[**QMESHARE**]**
When enqueuing a message from a specified queue, the specified queue is opened *exclusively* by another application. The other application is one based on a BEA product other than the BEA Tuxedo system that opened the queue for exclusive read and/or write using the Queuing Services API (QSAPI).

## See Also

qmadmin(1), gp_mktime(3c), tpacall(3c), tpalloc(3c), tpdequeue(3c), tpinit(3c), tpsprio(3c), APPQ_MIB(5), TMQFORWARD(5), TMQUEUE(5)

## tpenvelope(3c)

### Name

`tpenvelope()`—Accesses the digital signature and encryption information associated with a typed message buffer.

### Synopsis

```
#include <atmi.h>
int tpenvelope(char *data, long len, int occurrence, TPKEY *outputkey, long
*status, char *timestamp, long flags)
```

### Description

`tpenvelope()` provides access to the following types of digital signature and encryption information associated with a typed message buffer:

- Digital-signature registration requests

  A sending process *explicitly* registers a digital signature request for a message buffer by calling `tpsign()`, or *implicitly* registers a digital signature request for a message buffer by calling `tpkey_open()` with the `TPKEY_AUTOSIGN` flag specified.

- Digital signatures

  Just before the message buffer is sent, the public key software generates and attaches a digital signature to the message buffer for each digital-signature registration request; a digital signature enables a receiving process to verify the signer (originator) of the message.

- Encryption registration requests

  A sending process *explicitly* registers an encryption (seal) request for a message buffer by calling `tpseal()`, or *implicitly* registers an encryption (seal) request for a message buffer by calling `tpkey_open()` with the `TPKEY_AUTOENCRYPT` flag specified.

- Encryption envelopes

  Just before the message buffer is sent, the public key software encrypts the message content and attaches an encryption envelope to the message buffer for each encryption registration request; an encryption envelope enables a receiving process to decrypt the message.

Signature and encryption information is available to both sending and receiving processes. In a sending process, digital signature and encryption information is generally in a pending state,

waiting until the message is sent. In a receiving process, digital signatures have already been verified, and encryption and decryption have already been performed. Failures in decryption or signature verification might prevent message delivery, in which case the receiving process never receives the message buffer and therefore has no knowledge of the message buffer.

*data* must point to a valid typed message buffer either (1) previously allocated by a process calling tpalloc() or (2) delivered by the system to a receiving process. If the message buffer is self-describing, *len* is ignored (and may be 0). Otherwise, *len* must contain the length of data in *data*.

There may be multiple occurrences of digital-signature registration requests, digital signatures, encryption registration requests, and encryption envelopes associated with a message buffer. The occurrences are stored in sequence, with the first item at the zero position and subsequent items in consecutive positions. The *occurrence* input parameter indicates which item is requested. When the value of *occurrence* is beyond the position of the last item, tpenvelope() fails with the TPENOENT error condition. All items may be examined by calling tpenvelope() repeatedly until TPENOENT is returned.

The handle to the key associated with a digital-signature registration request, digital signature, encryption registration request, or encryption envelope is returned via *outputkey*. The key handle returned is a separate copy of the original key opened by calling tpkey_open(). Properties of the key, such as the PRINCIPAL attribute parameter, can be obtained by calling tpkey_getinfo(). It is the caller's responsibility to release key handle *outputkey* by calling tpkey_close().

**Note:** If *outputkey* is NULL, no key handle is returned.

The *status* output parameter reports the state of the digital-signature registration request, digital signature, encryption registration request, or encryption envelope. If the value of the status is not NULL, it is set to one of the following states:

TPSIGN_PENDING
: A digital signature has been requested on behalf of the *signer* principal associated with the corresponding private key, and will be generated when the message buffer is transmitted from this process.

TPSIGN_OK
: The digital signature has been verified.

TPSIGN_TAMPERED_MESSAGE
: The digital signature is not valid because the content of the message buffer has been altered.

TPSIGN_TAMPERED_CERT
: The digital signature is not valid because the signer's digital certificate has been altered.

TPSIGN_REVOKED_CERT
: The digital signature is not valid because the signer's digital certificate has been revoked.

TPSIGN_POSTDATED
: The digital signature is not valid because its timestamp is too far into the future.

TPSIGN_EXPIRED_CERT
: The digital signature is not valid because the signer's digital certificate has expired.

TPSIGN_EXPIRED
: The digital signature is not valid because its timestamp is too old.

TPSIGN_UNKNOWN
: The digital signature is not valid because the signer's digital certificate was issued by an unknown Certification Authority (CA).

TPSEAL_PENDING
: An encryption (seal) has been requested for the *recipient* principal associated with the corresponding public key, and will be performed when the message buffer is transmitted from this process.

TPSEAL_OK
: The encryption envelope is valid.

TPSEAL_TAMPERED_CERT
: The encryption envelope is not valid because the recipient's digital certificate has been altered.

TPSEAL_REVOKED_CERT
: The encryption envelope is not valid because the recipient's digital certificate has been revoked.

TPSEAL_EXPIRED_CERT
: The encryption envelope is not valid because the recipient's digital certificate has expired.

TPSEAL_UNKNOWN
: The encryption envelope is not valid because the recipient's digital certificate was issued by an unknown CA.

The *timestamp* output parameter contains the digital signature's timestamp according to the local clock on the machine where the digital signature was generated. The integrity of this value is protected by the associated digital signature. The memory location indicated by *timestamp* is set to the NULL-terminated signature time in format YYYYMMDDHHMMSS, where YYYY=year, MM=month, DD=day, HH=hour, MM=minute, and SS=second. *timestamp* may be NULL, in which

case no value is returned. Encryption seals do not contain timestamps, and the memory location indicated by *timestamp* is unchanged.

The *flags* parameter may be set to one of the following values:

- TPKEY_REMOVE–The item at position *occurrence* is removed (that is, it is no longer associated with the buffer). Output parameters *outputkey*, *status*, and *timestamp* related to the item are captured before the item is removed. Items at subsequent positions are shifted down by one, so there are never any gaps in the numbering of *occurrence*.
- TPKEY_REMOVEALL–All items associated with the message buffer are removed. The output parameters *outputkey*, *status*, and *timestamp* are not returned.
- TPKEY_VERIFY–All digital signatures associated with the message buffer are reverified. The status of a signature may change after reverification. For example, if a message buffer has been modified by a receiving process, the status of the originator's signature changes from TPSIGN_OK to TPSIGN_TAMPERED_MESSAGE.

### Return Values

On failure, this function returns -1 and sets tperrno to indicate the error condition.

### Errors

**[**TPEINVAL**]**
Invalid arguments were given. For example, the value of *data* is NULL or the value assigned to *flags* is unrecognized.

**[**TPENOENT**]**
This *occurrence* does not exist.

**[**TPESYSTEM**]**
An error occurred. Consult the system error log file for details.

### See Also

tpkey_close(3c), tpkey_getinfo(3c), tpkey_open(3c), tpseal(3c), tpsign(3c)

## tperrordetail(3c)

### Name

tperrordetail()—Gets additional detail about an error generated from the last BEA Tuxedo ATMI system call.

## Synopsis

```
#include <atmi.h>
int tperrordetail(long flags)
```

## Description

`tperrordetail()` returns additional detail related to an error produced by the last BEA Tuxedo ATMI system routine called in the current thread. `tperrordetail()` returns a numeric value that is also represented by a symbolic name. If the last BEA Tuxedo ATMI system routine called in the current thread did not produce an error, then `tperrordetail()` will return zero. Therefore, `tperrordetail()` should be called after an error has been indicated; that is, when `tperrno` has been set.

Currently *`flags`* is reserved for future use and must be set to 0.

A thread in a multithreaded application may issue a call to `tperrordetail()` while running in any context state, including `TPINVALIDCONTEXT`.

## Return Values

Upon failure, `tperrordetail()` returns a `-1` and sets `tperrno` to indicate the error condition.

These are the symbolic names and meaning for each numeric value that `tperrordetail()` may return. The order in which these are listed is not significant and does not imply precedence.

`TPED_CLIENTDISCONNECTED`

A Jolt client is disconnected currently. The `TPACK` flag is used in a `tpnotify()` call and the target of `tpnotify()` is a currently disconnected Jolt client. When `tpnotify()` fails, a subsequent call to `tperrordetail()` with no intermediate ATMI calls will return `TPED_CLIENTDISCONNECTED`.

`TPED_DECRYPTION_FAILURE`

A process receiving an encrypted message cannot decrypt the message. This error most likely occurs because the process does not have access to the private key required to decrypt the message.

When a call fails due to this error, a subsequent call to `tperrordetail()` with no intermediate ATMI calls will return `TPED_DECRYPTION_FAILURE`.

`TPED_DOMAINUNREACHABLE`

A domain is unreachable. Specifically, a domain configured to satisfy a request that a local domain cannot service was not reachable when a request was made. After the request failure, a subsequent call to `tperrordetail()` with no intermediate ATMI calls will return `TPED_DOMAINUNREACHABLE`.

The following table indicates the corresponding values returned by tperrno when calls to tpcall(), tpgetrply(), or tprecv() fail because of an unreachable domain. The error detail returned by a subsequent call to tperrordetail() is TPED_DOMAINUNREACHABLE.

| ATMI Call | tperrno | Error Detail |
|---|---|---|
| tpcall | TPESVCERR | TPED_DOMAINUNREACHABLE |
| tpgetrply | TPESVCERR | TPED_DOMAINUNREACHABLE |
| tprecv | TPEEVENT<br>TPEV_SVCERR | TPED_DOMAINUNREACHABLE |

Note that the TPED_DOMAINUNREACHABLE feature applies to BEA Tuxedo Domains only. It does not apply to other domains products such as Connect OSI TP Domains and Connect SNA Domains.

TPED_INVALID_CERTIFICATE

A process receiving a digitally signed message cannot verify the digital signature because the associated digital certificate is invalid. This error most likely occurs because the digital certificate has expired, the digital certificate was issued by an unknown Certification Authority (CA), or the digital certificate has been altered.

When a call fails due to this error, a subsequent call to tperrordetail() with no intermediate ATMI calls will return TPED_INVALID_CERTIFICATE.

TPED_INVALID_SIGNATURE

A process receiving a digitally signed message cannot verify the digital signature because the signature is invalid. This error most likely occurs because the message has been altered, the timestamp for the digital signature is too old, or the timestamp for the digital signature is too far into the future.

When a call fails due to this error, a subsequent call to tperrordetail() with no intermediate ATMI calls will return TPED_INVALID_SIGNATURE.

TPED_INVALIDCONTEXT

A thread is blocked in an ATMI call when another thread terminates its context. Specifically, any thread blocked in an ATMI call when another thread terminates its context will return from the ATMI call with a failure return; tperrno is set to TPESYSTEM. A subsequent call to tperrordetail() with no intermediate ATMI calls will return TPED_INVALIDCONTEXT.

TPED_INVALID_XA_TRANSACTION
: An attempt was made to start a transaction but the NO_XA flag was turned on in this domain.

TPED_NOCLIENT
: No client exists. The TPACK flag is used in a tpnotify() call but there is no target for tpnotify(). When tpnotify() fails, tperrno is set to TPENOENT. A subsequent call to tperrordetail() with no intermediate ATMI calls will return TPED_NOCLIENT.

TPED_NOUNSOLHANDLER
: A client does not have an unsolicited handler set. The TPACK flag is used in a tpnotify() call and the target of the tpnotify() is in a BEA Tuxedo ATMI session, but it has not set an unsolicited notification handler. When tpnotify() fails, tperrno is set to TPENOENT. A subsequent call to tperrordetail() with no intermediate ATMI calls will return TPED_NOUNSOLHANDLER.

TPED_SVCTIMEOUT
: A server was terminated due to a service timeout. The service timeout is controlled by the value of SVCTIMEOUT in the UBBCONFIG file or TA_SVCTIMEOUT in T_SERVER and T_SERVICE classes in the TM_MIB. When a call fails due to this error, a subsequent call to tperrordetail() with no intermediate ATMI calls will return TPED_SVCTIMEOUT.

TPED_TERM
: A Workstation client has been disconnected from the application. When a call fails due to this error, a subsequent call to tperrordetail() with no intermediate ATMI calls will return TPED_TERM.

### Errors

Upon failure, tperrordetail() sets tperrno to one of the following values:

TPEINVAL
: *flags* not set to zero

### See Also

Introduction to the C Language Application-to-Transaction Monitor Interface, tpstrerrordetail(3c), tperrno(5)

## tpexport(3c)

### Name

tpexport()—Converts a typed message buffer into an exportable, machine-independent string representation, that includes digital signatures and encryption envelopes.

## Synopsis

```
#include <atmi.h>
int tpexport(char *ibuf, long ilen, char *ostr, long *olen,
long flags)
```

## Description

`tpexport()` converts a typed message buffer into an externalized representation. An externalized representation is a message buffer that does *not* include any BEA Tuxedo ATMI header information that is normally added to a message buffer just before the buffer is transmitted.

The externalized representation may be transmitted between processes, machines, or BEA Tuxedo ATMI applications via any communication mechanism. It may be archived on permanent storage, and remains valid after a system shutdown and reboot.

An externalized representation includes:

- Any digital signatures associated with *ibuf*. They are verified later when the buffer is imported.
- Any encryption envelopes associated with *ibuf*. The buffer content remains protected by encryption. Only specified recipients with access to a valid private key for decryption may later import the buffer.

*ibuf* must point to a valid typed message buffer either (1) previously allocated by a process calling `tpalloc()` or (2) delivered by the system to a receiving process. *ilen* specifies how much of *ibuf* to export. Note that if *ibuf* points to a buffer type for which a length need not be specified (for example, an `FML` fielded buffer), then *ilen* is ignored (and may be 0).

*ostr* is a pointer to the output area that will hold an externalized representation of the buffer's content and associated properties. If `TPEX_STRING` is set in *flags*, then the externalized format will be a string type. Otherwise, the output length is determined by **olen* and may contain embedded NULL bytes.

On input, **olen* specifies the maximum storage size available at *ostr*. On output **olen* is set to the actual number of bytes written to *ostr* (including a terminating NULL character if `TPEX_STRING` is set in *flags*).

The *flags* argument may be set to `TPEX_STRING` if string format (base 64 encoded) is desired for the output buffer. Otherwise, the output will be binary.

### Return Values

On failure, this function returns -1 and sets `tperrno` to indicate the error condition.

### Errors

**[**`TPEINVAL`**]**
Invalid arguments were given. For example, the value of *`ibuf`* is NULL or the value of *`flags`* is not set correctly.

**[**`TPEPERM`**]**
Permission failure. The cryptographic service provider was not able to access a private key necessary to produce a digital signature.

**[**`TPESYSTEM`**]**
An error occurred. Consult the system error log file for details.

**[**`TPELIMIT`**]**
Insufficient output storage was provided. ***`olen`* is set to the necessary amount of space.

### See Also

`tpimport(3c)`

## tpfml32toxml(3c)

### Name

`tpfml32toxml()`—Converts FML32 buffers to XML data

### Synopsis

```
#include <fml32.h>
int tpfml32toxml (FBFR32 *fml32bufp, char *vfile, char *rtag, char
**xmlbufp, long flags)
```

### Description

This function is used to convert FML32 buffers to XML data. It supports the following valid arguments:

`fml32bufp`
This argument is a pointer to an input FML32 typed buffer.

`vfile`
: This argument is not used for FML32 to XML conversion at this time. It is reserved for the fully qualified path name of an XML Schema file used to validate XML output when this capability is supported by Xerces.

`rtag`
: The argument is a pointer to the input root element name for the output XML document.When a root element name is specified during conversion, it is identified and saved for use as an XML root tag with an optional `Type` attribute added to the root element name. If the input root name is not specified, then the default output XML root tag <FML32> is used.

`xmlbufp`
: This argument is a pointer to an output XML typed buffer in a pre-defined format for describing FML32 fielded buffers.

`flag`
: This argument is not used for FML32 to XML conversion at this time and should be set to 0.

## Return Values

Upon success, `tpfml32toxml()` returns `0`. This function returns `-1` on error and sets `tperrno` to indicate the error condition.

## Errors

Upon failure, `tpfml32toxml()` sets `tperrno` to one of the following values:

**[**`TPEINVAL`**]**
: Either `fml32bufp` or `xmlbufp` is not a valid typed buffer.

**[**`TPESYSTEM`**]**
: A Tuxedo system error has occurred. The exact nature of the error is written to `userlog(3)`. This will also indicate when a conversion to XML was unable to be done. In that instance error detail info will be added to the userlog.

**[**`TPEOS`**]**
: An operating system error has occurred. A numeric value representing the system call that failed is available in Uunixerr.

## SEE ALSO

`tpxmltofml32(3c), tpxmltofml(3c), tpfmltoxml(3c)`

# tpfmltoxml(3c)

## Name

`tpfmltoxml()`—Converts FML buffers to XML data

## Synopsis

```
#include <fml.h>
int tpfmltoxml (FBFR *fmlbufp, char *vfile, char *rtag, char **xmlbufp, long
flags)
```

## Description

This function is used to convert FML buffers to XML data. It supports the following valid arguments:

`fmlbufp`
: The argument is a pointer to an input FML typed buffer.

`vfile`
: The argument is not used for FML to XML conversion at this time. It is reserved for the fully qualified path name of an XML Schema file used to validate XML output when this capability is supported by Xerces.

`rtag`
: This argument is a pointer to the input root element name for the output XML document.When a root element name is specified during conversion, it is identified and saved for use as an XML root tag with an optional `Type` attribute added to the root element name. If the input root name is not specified, then the default output XML root tag `<FML>` is used.

`xmlbufp`
: This argument is a pointer to an output XML typed buffer in a pre-defined format for describing FML fielded buffers.

`flag`
: This argument is not used for FML to XML conversion at this time and should be set to `0`.

## Return Values

Upon success, `tpfmltoxml()` returns `0`. This function returns `-1` on error and sets `tperrno` to indicate the error condition.

### Errors

Upon failure, `tpfmltoxml()` sets `tperrno` to one of the following values:

**[**`TPEINVAL`**]**
Either `fml32bufp` or `xmlbufp` is not a valid typed buffer.

**[**`TPESYSTEM`**]**
A Tuxedo system error has occurred. The exact nature of the error is written to `userlog(3)`. This will also indicate when a conversion to XML was unable to be done. In that instance error detail info will be added to the userlog.

`[TPEOS]`
An operating system error has occurred. A numeric value representing the system call that failed is available in Uunixerr.

### SEE ALSO

`tpxmltofml32(3c),tpfml32toxml(3c), tpxmltofml(3c)`

## tpforward(3c)

### Name

`tpforward()`—Routine for forwarding a service request to another service routine.

### Synopsis

```
#include <atmi.h>
void tpforward(char *svc, char *data, long len, long flags)
```

### Description

`tpforward()` allows a service routine to forward a client's request to another service routine for further processing. `tpforward()` acts like `tpreturn()` in that it is the last call made in a service routine. Like `tpreturn()`, `tpforward()` should be called from within the service routine dispatched to ensure correct return of control to the BEA Tuxedo ATMI system dispatcher. `tpforward()` cannot be called from within a conversational service.

This function forwards a request to the service named by *`svc`* using data pointed to by *`data`*. The service name must not begin with a dot. A service routine forwarding a request receives no reply. After the request is forwarded, the service routine returns to the communication manager dispatcher and the server is free to do other work. Note that because no reply is expected from a

forwarded request, the request may be forwarded without error to any service routine in the same executable as the service that forwarded the request.

If the service routine is in transaction mode, `tpforward()` puts the caller's portion of the transaction in a state where it may be completed when the originator of the transaction issues either `tpcommit()` or `tpabort()`. If a transaction was explicitly started with `tpbegin()` while in a service routine, the transaction must be ended with either `tpcommit()` or `tpabort()` before calling `tpforward()`. Thus, all services in a "forward chain" are either all started in transaction mode or none are.

The last server in a forward chain sends a reply back to the originator of the request using `tpreturn()`. In essence, `tpforward()` transfers to another server the responsibility of sending a reply back to the awaiting requester.

`tpforward()` should be called after receiving all replies expected from service requests initiated by the service routine. Any outstanding replies which are not received will automatically be dropped by the communication manager dispatcher upon receipt. In addition, the descriptors for those replies become invalid and the request is not forwarded to *svc*.

*data* points to the data portion of a reply to be sent. If *data* is non-NULL, it must point to a buffer previously obtained by a call to `tpalloc()`. If this is the same buffer passed to the service routine upon its invocation, then its disposition is up to the BEA Tuxedo ATMI system dispatcher; the service routine writer does not have to worry about whether it is freed or not. In fact, any attempt by the user to free this buffer will fail. However, if the buffer passed to `tpforward()` is not the same one with which the service is invoked, then `tpforward()` will free that buffer. *len* specifies the amount of the data buffer to be sent. If *data* points to a buffer which does not require a length to be specified, (for example, an FML fielded buffer), then *len* is ignored (and can be 0). If *data* is NULL, then *len* is ignored and a request with zero length data is sent.

The *flags* argument is reserved for future use and should be set to 0 (zero) for Tuxedo 9.1 rolling patches earlier than RP073. For RP073 and later rolling patches, the flags argument can be set to 0 (zero) or `TPENOBLOCK`.

When 0 (zero) is specified and a blocking condition exists, the call is blocked until the condition subsides or a timeout occurs (for example, a transaction or blocking timeout). When `TPENOBLOCK` is specified and a blocking condition exists, the call is returned immediately with `TPESVCERR`.

## Return Values

A service routine does not return any value to its caller, the communication manager dispatcher. Thus, `tpforward()` is declared as a void. See `tpreturn(3c)` for a more extensive discussion.

### Errors

If any errors occur either in the handling of the parameters passed to the function or in its processing, a "failed" message is sent back to the original requester (unless no reply is to be sent). The existence of outstanding replies or subordinate connections, or the caller's transaction being marked abort-only, qualify as failures which generate failed messages.

If either SVCTIMEOUT in the UBBCONFIG file or TA_SVCTIMEOUT in the TM_MIB is non-zero, the event, TPEV_SVCERR is returned when a service timeout occurs.

Failed messages are detected by the requester with the TPESVCERR error indication. When such an error occurs, the caller's data is not sent. Also, this error causes the caller's current transaction to be marked abort-only.

If a transaction timeout occurs, either during the service routine or while the request is being forwarded, the requester waiting for a reply with either tpcall() or tpgetrply() will get a TPETIME error return. When a service fails inside a transaction, the transaction times out and is put into the TX_ROLLBACK_ONLY state. All further ATMI calls for that transaction will fail with TPETIME. The waiting requester will not receive any data. Service routines, however, are expected to terminate using either tpreturn() or tpforward(). A conversational service routine must use tpreturn(); it cannot use tpforward().

If a service routine returns without using either tpreturn() or tpforward() (that is, if it uses the C language return statement or simply "falls out of the function") or if tpforward() is called from a conversational server, the server will print a warning message in a log file and return a service error to the original requester. All open connections to subordinates will be disconnected immediately, and any outstanding asynchronous replies will be marked stale. If the server was in transaction mode at the time of failure, the transaction is marked abort-only. Note also that if either tpreturn() or tpforward() are used outside of a service routine (for example, in clients, or in tpsvrinit() or tpsvrdone()), then these routines simply return having no effect.

### See Also

tpalloc(3c), tpconnect(3c), tpreturn(3c), tpservice(3c), tpstrerrordetail(3c)

## tpfree(3c)

### Name

tpfree()—Routine for freeing a typed buffer.

## Synopsis

```
#include <atmi.h>
void tpfree(char *ptr)
```

## Description

The argument to `tpfree()` is a pointer to a buffer previously obtained by either `tpalloc()` or `tprealloc()`. If *`ptr`* is NULL, no action occurs. Undefined results will occur if *`ptr`* does not point to a typed buffer (or if it points to space previously freed with `tpfree()`). Inside service routines, `tpfree()` returns and does not free the buffer if *`ptr`* points to the buffer passed into a service routine.

Some buffer types require state information or associated data to be removed as part of freeing a buffer. `tpfree()` removes any of these associations (in a communication manager-specific manner) before a buffer is freed.

Once `tpfree()` returns, *`ptr`* should not be passed as an argument to any BEA Tuxedo ATMI system routine or used in any other manner.

A thread in a multithreaded application may issue a call to `tpfree()` while running in any context state, including `TPINVALIDCONTEXT`.

When freeing an `FML32` buffer using `tpfree()`, the routine recursively frees all embedded buffers to prevent memory leaks. In order to preserve the embedded buffers, you should assign the associated pointer to NULL before issuing the `tpfree()` command. As stated above, if *ptr* is NULL, no action occurs.

## Return Values

`tpfree()` does not return any value to its caller. Thus, it is declared as a void.

## Usage

This function should not be used in concert with `malloc()`, `realloc()`, or `free()` in the C library (for example, a buffer allocated with `tpalloc()` should not be freed with `free()`).

## See Also

Introduction to the C Language Application-to-Transaction Monitor Interface, `tpalloc(3c)`, `tprealloc(3c)`

# tpgblktime(3c)

## Name

tpgblktime()—Retrieves a previously set, per second, blocktime value

## Synopsis

```
#include <atmi.h>
int tpgblktime(TPBLK_NEXT, TPBLK_ALL long flags)
```

## Description

tpgblktime() retrieves a previously set, per second, blocktime value. If tpgblktime() specifies a blocktime flag value, and no such flag value has been set, the return value is 0. A blocktime flag value less than 0 produces an error.

The following is a list of valid flags:

TPBLK_NEXT
: This flag returns the per second blocktime value for the previously set tpsblktime(TPBLK_NEXT) call.

TPBLK_ALL
: This flag returns the per second blocktime value for the previously set tpsblktime(TPBLK_ALL) call.

0
: This flag returns the applicable blocktime value for the next blocking ATMI set due to a previous tpsblktime() call with the TPBLK_NEXT or TPBLK_ALL flag blocktime value, or a system-wide default blocktime value.

**Note:** When a workstation client calls a tpgblktime() 0 flag, the system-wide default blocktime value cannot be returned. A 0 value is returned instead.

## Return Values

Upon success, tpgblktime() returns a positive integer indicating the blocking time value currently in effect for the corresponding flag value. A 0 return value indicates that no such blocking time override is currently in effect.

This function returns -1 on error and sets tperrno to indicate the error condition. The failure does not affect the existing transaction, if one exists.

## Error

Upon failure, tpgblktime() sets tperrno to one of the following values:

**[**TPEINVAL**]**

Invalid arguments were given. For example, the *flags* value is negative or more than one blocktime flag value (TPBLK_NEXT, TPBLK_ALL, or 0) was specified.

**[**TPESYSTEM**]**

A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

### See Also

tpcall(3c), tpcommit(3c), tprecv(3c), tpsblktime(3c), UBBCONFIG(5)

## tpgetadmkey(3c)

### Name

tpgetadmkey()—Gets administrative authentication key.

### Synopsis

```
#include <atmi.h>
long tpgetadmkey(TPINIT *tpinfo)
```

### Description

tpgetadmkey() is available for application use by an application specific authentication server. It returns an application security key suitable for assignment to the indicated user for the purpose of administrative authentication. This routine must be called with a client name (that is, *tpinfo->cltname*) of either tpsysadm() or tpsysop(); otherwise, a valid administrative key will not be returned.

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tpgetadmkey().

### Return Values

Upon success, tpgetadmkey() returns a non-0 value with the high-order bit (0x80000000) set; otherwise it returns 0. Zero may be returned if *tpinfo* is NULL, *tpinfo->cltname* is not tpsysadm() or tpsysop(), or lastly if the effective user ID is not the configured application administrator for this site.

### Errors

A zero return value is the only indication that a valid administrative key was not assigned.

### Portability

This interface is available only on UNIX system sites running BEA Tuxedo release 5.0 or later.

### See Also

`tpaddusr(1)`, `tpusradd(1)`, `tpinit(3c)`, `AUTHSVR(5)`

*Setting Up a BEA Tuxedo Application*

*Administering a BEA Tuxedo Application at Run Time*

## tpgetctxt(3c)

### Name

`tpgetctxt()`—Retrieves a context identifier for the current application association.

### Synopsis

```
#include <atmi.h>
int tpgetctxt(TPCONTEXT_T *context, long flags)
```

### Description

`tpgetctxt()` retrieves an identifier that represents the current application context and places that identifier in *`context`*. This function operates on a per-thread basis in a multithreaded environment, and on a per-process basis in a non-threaded environment.

Typically, a thread:

1. Calls `tpinit()`
2. Calls `tpgetctxt()`
3. Handles the value of *`context`* as follows:
   - In a multithreaded application—passes the value of *`context`* to another thread in the same process so the other thread can call `tpsetctxt()`.
   - In a single-threaded or multithreaded application—saves this context identifier for itself so it can switch back to the indicated context later.

The second argument, *`flags`*, is not currently used and must be set to 0.

`tpgetctxt()` may be called in single-context applications as well as in multicontext applications.

A thread in a multithreaded application may issue a call to tpgetctxt() while running in any context state, including TPINVALIDCONTEXT.

## Return Values

Upon successful completion, tpgetctxt() returns a non-negative value. Context is set to the current context ID, which may be represented by any of the following:

- A context ID greater than 0, indicating a context in a multicontexted application.
- TPSINGLECONTEXT, indicating that the current thread has successfully executed tpinit() without the TPMULTICONTEXTS flag, or that the current thread was just created in a process that has successfully executed tpinit() without the TPMULTICONTEXTS flag. The value of TPSINGLECONTEXT is 0.
- TPNULLCONTEXT, indicating that the current thread is not associated with a context.
- TPINVALIDCONTEXT, indicating that the current thread is in the invalid context state. If a thread in a multicontexted client issues a call to tpterm() while other threads in the same context are still working, the working threads are placed in the TPINVALIDCONTEXT context. The value of TPINVALIDCONTEXT is -1.

  A thread in the TPINVALIDCONTEXT state is prohibited from issuing calls to most ATMI functions. For a complete list of functions that may and may not be called, see the Introduction to the C Language Application-to-Transaction Monitor Interface.

  For details about the TPINVALIDCONTEXT context state, see tpterm(3c).

Upon failure, tpgetctxt() returns a value of -1 and sets tperrno to indicate the error condition.

## Errors

Upon failure, tpgetctxt() sets tperrno to one of the following values:

**[**TPEINVAL**]**
: Invalid arguments have been given. For example, the value of *context* is NULL or the value of *flags* is not 0.

**[**TPESYSTEM**]**
: A BEA Tuxedo system error has occurred. The exact nature of the error has been written to a log file.

**[**TPEOS**]**
: An operating system error has occurred.

### See Also

Introduction to the C Language Application-to-Transaction Monitor Interface, `tpsetctxt(3c)`, `tpterm(3c)`

## tpgetlev(3c)

### Name

`tpgetlev()`—Routine for checking if a transaction is in progress.

### Synopsis

```
#include <atmi.h>
int tpgetlev()
```

### Description

`tpgetlev()` returns to the caller the current transaction level. Currently, the only levels defined are 0 and 1.

In a multithreaded application, a thread in the `TPINVALIDCONTEXT` state is not allowed to issue a call to `tpgetlev()`.

### Return Values

Upon successful completion, `tpgetlev()` returns either a 0 to indicate that no transaction is in progress, or 1 to indicate that a transaction is in progress;

Upon failure, `tpgetlev()` returns -1 and sets `tperrno` to indicate the error condition.

### Errors

Upon failure, `tpgetlev()` sets `tperrno` to one of the following values:

**[**`TPEPROTO`**]**
`tpgetlev()` was called improperly.

**[**`TPESYSTEM`**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**`TPEOS`**]**
An operating system error has occurred.

### Notices

When using `tpbegin()`, `tpcommit()` and `tpabort()` to delineate a BEA Tuxedo ATMI system transaction, it is important to remember that only the work done by a resource manager that meets the XA interface (and is linked to the caller appropriately) has transactional properties. All other operations performed in a transaction are not affected by either `tpcommit()` or `tpabort()`. See `buildserver(1)` for details on linking resource managers that meet the XA interface into a server such that operations performed by that resource manager are part of a BEA Tuxedo ATMI system transaction.

### See Also

`tpabort(3c)`, `tpbegin(3c)`, `tpcommit(3c)`, `tpscmt(3c)`

## tpgetmbenc(3c)

### Name

`tpgetmbenc()`—Gets the code-set encoding name from a typed buffer.

### Synopsis

```
#include <atmi.h>
extern int tperrno;
int
tpgetmbenc (char *bufp, char *enc_name, long flags)
```

### Description

This function is used to get the codeset encoding name sent with a typed buffer. This name can be compared to a target codeset if a conversion is required (see tpconvmb(3c)).

The *`bufp`* argument is a valid pointer to a typed buffer message.

The *`enc_name`* argument will be set to the encoding name, found in *`bufp`*, upon successful execution of this function. The returned string will be NULL terminated. The user must take care to allocate a buffer large enough to hold the encoding name plus the NULL terminator (see `NL_LANGMAX` in `<limits.h>`). An MBSTRING typed buffer without the encoding name set is invalid.

The *`flags`* argument is not currently used and should be set to zero.

### Return Values

Upon success, tpgetmbenc() returns a value of 0. This function returns -1 on error and sets tperrno as described below for each function. The function may fail for the following reasons.

**[**TPEINVAL**]**
*enc_name* or *bufp* argument is NULL.

**[**TPEPROTO**]**
This error occurs if *bufp* cannot provide an encoding name.

**[**TPESYSTEM**]**
A Tuxedo system error has occurred. (e.g. *bufp* does not correspond to a valid Tuxedo buffer).

### See Also

tpalloc(3c), tpconvmb(3c), tpsetmbenc(3c)

## tpgetrepos(3c)

### Name

tpgetrepos() - retrieves service parameter information from a Tuxedo service metadata repository file.

```
#include <atmi.h>
```

### Synopsis

```
int tpgetrepos(char *reposfile, FBFR32* idata, FBFR32** odata)
```

### Description

tpgetrepos() provides an alternative repository access interface to the .TMMETAREPOS service provided by TMMETADATA(5). It retrieves service parameters from a Tuxedo service metadata repository file. To use tpgetrepos(), the metadata repository file must reside on the native client or server that initiates the request. This allows for repository information access even when TMMETADATA(5) has not been booted.

**Note:** tpgetrepos() can also be used to view Jolt repository files. It cannot modify an existing Jolt repository file or create a new one.

tpgetrepos() accepts the following parameters:

reposfile
: specifies the path name of a file accessible on the current machine where the Tuxedo Metadata Repository is located. The caller must have read permission for this file.

idata
: specifies what type of service parameter information is retrieved, and points to an FML32 buffer.

*odata
: On output, points to an FML32 buffer containing the retrieved service parameter information and operation status.

METAREPOS(5) describes the FML32 buffer format tpgetrepos()uses. It is similar to the format used by the Tuxedo MIB.

## Return Value

tpgetrepos() returns 0 on success. On failure, it sets tperrno and returns -1. On most failure conditions, the TA_ERROR field in *odata is populated with information about the specific error, as is done by the Tuxedo MIB.

## Errors

Upon failure, tpgetrepos() sets tperrno to one of the following values:

**Note:** Except for TPEINVAL, odata is modified to include TA_ERROR, TA_STATUS for each service entry to further qualify the error condition.

[TPEINVAL]
: Invalid arguments were specified. The reposfile value is invalid or idata or odata are not pointers to FML32 typed buffers.

[TPEMIB]
: The MIB-like request failed. odata is updated and returned to the caller with FML32 fields indicating the cause of the error as discussed in MIB(5).

[TPEPROTO]
: tpgetrepos() was improperly called. The reposfile file argument given is not a valid repository file.

[TPEOS]
: An operating system error has occurred. A numeric value representing the system call that failed is available in Uunixerr.

[TPESYSTEM]
: A BEA Tuxedo system error has occurred. The exact nature of the error is reported in userlog().

### Portability

This interface is available only on BEA Tuxedo release 9.0 or later.

### Files

The following library files are required:

```
${TUXDIR}/lib/libtrep.a
${TUXDIR}/lib/libtrep.so.<rel>
${TUXDIR}/lib/libtrep.lib
```

The libraries must be linked manually when using buildclient. The user must use:
`-L${TUXDIR}/lib -ltrep`

### See Also

`tpsetrepos(3c)`, `tmloadrepos(1)`, `tmunloadrepos(1)`, `TMMETADATA(5)`, Managing The Tuxedo Service Metadata Repository

## tpgetrply(3c)

### Name

`tpgetrply()`—Routine for getting a reply from a previous request.

### Synopsis

```
#include <atmi.h>
int tpgetrply(int *cd, char **data, long *len, long flags)
```

### Description

`tpgetrply()` returns a reply from a previously sent request. This function's first argument, *cd*, points to a call descriptor returned by `tpacall()`. By default, the function waits until the reply matching **cd* arrives or a timeout occurs.

*data* must be the address of a pointer to a buffer previously allocated by `tpalloc()` and *len* should point to a long that `tpgetrply()` sets to the amount of data successfully received. Upon successful return, **data* points to a buffer containing the reply and **len* contains the size of the data. FML and FML32 buffers often assume a minimum size of 4096 bytes; if the reply is larger than 4096, the size of the buffer is increased to a size large enough to accommodate the data being returned. As of release 6.4, the default allocation for buffers is 1024 bytes. Also, historical information is maintained on recently used buffers, allowing a buffer of optimal size to be reused as a return buffer.

Buffers on the sending side that may be only partially filled (for example, FML or STRING buffers) will have only the amount that is used send. The system may then enlarge the received data size by some arbitrary amount. This means that the receiver may receive a buffer that is smaller than what was originally allocated by the sender, yet larger than the data that was sent.

The receive buffer may grow, or it may shrink, and its address almost invariably changes, as the system swaps buffers around internally. To determine whether (and how much) a reply buffer changed in size, compare its total size before `tpgetrply()` was issued with *`len`. See the "Introduction to the C Language Application-to-Transaction Monitor Interface" for more information about buffer management.

If *`len` is 0, then the reply has no data portion and neither *`data` nor the buffer it points to were modified.

It is an error for *`data` or `len` to be NULL.

Within any particular context of a multithreaded program:

- Calls to `tpgetrply(TPGETANY)` and `tpgetrply()` for a specific handle cannot be issued concurrently.
- Multiple calls to `tpgetrply(TPGETANY)` cannot be issued concurrently.

Any `tpgetrply()` call that would, if issued, cause a violation of either of these restrictions, returns -1 and sets `tperrno` to `TPEPROTO`.

It is acceptable to issue:

- Concurrent calls to `tpgetrply()` for different handles.
- A call to `tpgetrply`(`TPGETANY)` in a single context concurrently with a call to `tpgetrply()`, with or without `TPGETANY`, in a different context.

The following is a list of valid *`flags`*:

`TPGETANY`
: This flag signifies that `tpgetrply()` should ignore the descriptor pointed to by *`cd`*, return any reply available and set *`cd`* to point to the call descriptor for the reply returned. If no replies exist, `tpgetrply()` by default will wait for one to arrive.

`TPNOCHANGE`
: By default, if a buffer is received that differs in type from the buffer pointed to by *`data`*, then *`data`*'s buffer type changes to the received buffer's type so long as the receiver recognizes the incoming buffer type. When this flag is set, the type of the buffer pointed to by *`data`* is not allowed to change. That is, the type and subtype of the received buffer must match the type and subtype of the buffer pointed to by *`data`*.

TPNOBLOCK
: tpgetrply() does not wait for the reply to arrive. If the reply is available, then tpgetrply() gets the reply and returns. When this flag is not specified and a reply is not available, the caller blocks until the reply arrives or a timeout occurs (either transaction or blocking timeout).

TPNOTIME
: This flag signifies that the caller is willing to block indefinitely for its reply and wants to be immune to blocking timeouts. Transaction timeouts may still occur.

TPSIGRSTRT
: If a signal interrupts any underlying system calls, then the interrupted system call is reissued.

Except as noted below, **cd* is no longer valid after its reply is received.

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tpgetrply().

## Return Values

Upon successful return from tpgetrply() or upon return where tperrno is set to TPESVCFAIL, tpurcode() contains an application defined value that was sent as part of tpreturn().

Upon failure, tpgetrply() returns -1 and sets tperrno to indicate the error condition.

## Errors

Upon failure, tpgetrply() sets tperrno as indicated below. Note that if TPGETANY is not set, then **cd* is invalidated unless otherwise stated. If TPGETANY is set, then *cd* points to the descriptor for the reply on which the failure occurred; if an error occurred before a reply could be retrieved, then *cd* points to 0. Also, the failure does not affect the caller's transaction, if one exists, unless otherwise stated. If a call fails with a particular tperrno value, a subsequent call to tperrordetail() with no intermediate ATMI calls, may provide more detailed information about the generated error. Refer to the tperrordetail(3c) reference page for more information.

**[**TPEINVAL**]**
: Invalid arguments were given (for example, *cd*, *data*, **data* or *len* is NULL or *flags* are invalid). If *cd* is non-NULL, then it is still valid after this error and the reply remains outstanding.

**[**TPEOTYPE**]**
: Either the type and subtype of the reply are not known to the caller; or, TPNOCHANGE was set in *flags* and the type and subtype of **data* do not match the type and subtype of the

reply sent by the service. Regardless, neither *_data_, its contents nor *_len_ are changed. If the reply was to be received on behalf of the caller's current transaction, then the transaction is marked abort-only since the reply is discarded.

**[**TPEBADDESC**]**

_cd_ points to an invalid descriptor.

**[**TPETIME**]**

This error code indicates that either a timeout has occurred or `tpgetrply()` has been attempted, in spite of the fact that the current transaction is already marked rollback only.

If the caller is in transaction mode, then either the transaction is already rollback only or a transaction timeout has occurred. The transaction is marked abort-only. If the caller is not in transaction mode, a blocking timeout has occurred. (A blocking timeout cannot occur if `TPNOBLOCK` and/or `TPNOTIME` is specified.) In either case, no changes are made to `*data`, its contents, or `*len`. `*cd` remains valid unless the caller is in transaction mode (and `TPGETANY` has not been set).

If a transaction timeout has occurred, then, with one exception, any attempts to send new requests or receive outstanding replies will fail with `TPETIME` until the transaction has been aborted. The exception is a request that does not block, expects no reply, and is not sent on behalf of the caller's transaction (that is, `tpacall()` with `TPNOTRAN`, `TPNOBLOCK`, and `TPNOREPLY` set).

When a service fails inside a transaction, the transaction is put into the `TX_ROLLBACK_ONLY` state. This state is treated, for most purposes, as though it were equivalent to a timeout. All further ATMI calls for this transaction (with the exception of those issued in the circumstances described in the previous paragraph) will fail with `TPETIME`.

**[**TPESVCFAIL**]**

The service routine sending the caller's reply called `tpreturn()` with `TPFAIL`. This is an application-level failure. The contents of the service's reply, if one was sent, is available in the buffer pointed to by *_data_. If the service request was made on behalf of the caller's transaction, then the transaction is marked abort-only. Note that regardless of whether the transaction has timed out, the only valid communications before the transaction is aborted are calls to `tpacall()` with `TPNOREPLY`, `TPNOTRAN`, and `TPNOBLOCK` set.

**[**TPESVCERR**]**

A service routine encountered an error either in `tpreturn()` or `tpforward()` (for example, bad arguments were passed). No reply data is returned when this error occurs (that is, neither *_data_, its contents nor *_len_ are changed). If the service request was made on behalf of the caller's transaction, then the transaction is marked abort-only. Note that

regardless of whether the transaction has timed out, the only valid communications before the transaction is aborted are calls to `tpacall()` with `TPNOREPLY`, `TPNOTRAN`, and `TPNOBLOCK` set. If either `SVCTIMEOUT` in the `UBBCONFIG` file or `TA_SVCTIMEOUT` in the `TM_MIB` is non-zero, `TPESVCERR` is returned when a service timeout occurs.

**[**`TPEBLOCK`**]**
A blocking condition exists and `TPNOBLOCK` was specified. **cd* remains valid.

**[**`TPGOTSIG`**]**
A signal was received and `TPSIGRSTRT` was not specified.

**[**`TPEPROTO`**]**
`tpgetrply()` was called improperly.

**[**`TPESYSTEM`**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**`TPEOS`**]**
An operating system error has occurred. If a message queue on a remote location is filled, `TPEOS` may possibly be returned.

### See Also

`tpacall(3c)`, `tpalloc(3c)`, `tpcancel(3c)`, `tperrordetail(3c)`, `tprealloc(3c)`, `tpreturn(3c)`, `tpstrerrordetail(3c)`, `tptypes(3c)`

## tpgprio(3c)

### Name

`tpgprio()`—Routine for getting a service request priority.

### Synopsis

```
#include <atmi.h>
int tpgprio(void)
```

### Description

`tpgprio()` returns the priority for the last request sent or received by the current thread in its current context. Priorities can range from 1 to 100, inclusive, with 100 being the highest priority. `tpgprio()` may be called after `tpcall()` or `tpacall()`, (also `tpenqueue()`, or `tpdequeue()`, assuming the queued management facility is installed), and the priority returned is for the request

sent. Also, tpgprio() may be called within a service routine to find out at what priority the invoked service was sent. tpgprio() may be called any number of times and will return the same value until the next request is sent.

In a multithreaded application tpgprio() operates on a per-thread basis.

Because the conversation primitives are not associated with priorities, issuing tpsend() or tprecv() has no affect on the priority returned by tpgprio(). Also, there is no priority associated with a conversational service routine unless a tpcall() or tpacall() is done within that service.

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tpgprio().

## Return Values

Upon success, tpgprio() returns a request's priority;

Upon failure, tpgprio() returns -1 and sets tperrno to indicate the error condition.

## Errors

Upon failure, tpgprio() sets tperrno to one of the following values:

**[**TPENOENT**]**
: tpgprio() was called and no requests (via tpcall() or tpacall()) have been sent, or it is called within a conversational service for which no requests have been sent.

**[**TPEPROTO**]**
: tpgprio() was called improperly.

**[**TPESYSTEM**]**
: A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPEOS**]**
: An operating system error has occurred.

## See Also

tpacall(3c), tpcall(3c), tpdequeue(3c), tpenqueue(3c), tpservice(3c), tpsprio(3c)

# tpimport(3c)

## Name

tpimport()—Converts an externalized representation of a message buffer into a typed message buffer.

## Synopsis

```
#include <atmi.h>
int tpimport(char *istr, long ilen, char **obuf, long *olen,
long flags)
```

## Description

tpimport() converts an externalized representation of a message buffer into a typed message buffer. An externalized representation is a message buffer that does *not* include any BEA Tuxedo ATMI header information that is normally added to a message buffer just before the buffer is transmitted. A process converts a typed message buffer into an externalized representation by calling the tpexport() function.

Any digital signatures associated with *istr* are verified when the buffer is imported, and are available for examination after importing via tpenvelope().

If the *istr* buffer representation is encrypted, the importing process must have access to a valid private key for decryption. Decryption is performed automatically during the importing process.

If TPEX_STRING is *not* set in *flags*, then *ilen* contains the length of the binary data contained in *istr*. If *ilen* is 0, *istr* is assumed to point to a NULL-terminated string, and the TPEX_STRING flag is inferred.

**obuf* must point to a valid typed message buffer either (1) previously allocated by a process calling tpalloc() or (2) delivered by the system to a receiving process. The buffer will be reallocated as necessary to accommodate the result, and its buffer type or subtype may change.

**olen* is set to the amount of valid data contained in the output buffer. If *olen* is NULL on input, it is ignored.

The *flags* argument should be set to TPEX_STRING if the input externalized representation is in string format (base 64 encoded). Otherwise, the input is in binary format of length *ilen*.

## Return Values

On failure, this function returns -1 and sets tperrno to indicate the error condition.

### Errors

**[**TPEINVAL**]**
Invalid arguments were given. For example, the value of *istr* is NULL or the *flags* parameter is not set correctly.

**[**TPEPERM**]**
Permission failure. The cryptographic service provider was not able to access a private key necessary for decryption.

**[**TPEPROTO**]**
A protocol failure occurred. The failure involves an invalid data format in *istr* or a digital signature that failed verification.

**[**TPESYSTEM**]**
An error occurred. Consult the system error log file for more details.

### See Also

tpenvelope(3c), tpexport(3c)

## tpinit(3c)

### Name

tpinit()—Joins an application.

### Synopsis

```
#include <atmi.h>
int tpinit(TPINIT *tpinfo)
```

### Description

tpinit() allows a client to join a BEA Tuxedo ATMI system application. Before a client can use any of the BEA Tuxedo ATMI system communication or transaction routines, it must first join a BEA Tuxedo ATMI system application.

tpinit() has two modes of operation: single-context mode and multicontext mode, which will be discussed in detail below. Because calling tpinit() is optional when in single-context mode, a single-context client may also join an application by calling many ATMI routines (for example, tpcall()), which transparently call tpinit() with *tpinfo* set to NULL. A client may want to call tpinit() directly so that it can set the parameters described below. In addition, tpinit() must be used when multicontext mode is required, when application authentication is required

(see the description of the SECURITY keyword in UBBCONFIG(5)), or when the application wishes to supply its own buffer type switch (see typesw(5)). After tpinit() successfully returns, the client can initiate service requests and define transactions.

In single-context mode, if tpinit() is called more than once (that is, if it is called after the client has already joined the application), no action is taken and success is returned.

In a multithreaded client, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tpinit(). To join a BEA Tuxedo ATMI application, a multithreaded Workstation client must always call tpinit() with the TPMULTICONTEXTS flag set, even if the client is running in single-context mode.

**Note:** The TPMULTICONTEXTS mode of tpinit will continue to work properly when the TMNOTHREADS environment variable is set to yes. Setting this environment variable to yes turns off multithreaded processing for applications that do not use threads.

## Description of the TPINFO Structure

tpinit()'s argument, *tpinfo*, is a pointer to a typed buffer of type TPINIT and a NULL subtype. TPINIT is a buffer type that is typedefed in the atmi.h header file. The buffer must be allocated via tpalloc() prior to calling tpinit(). The buffer should be freed using tpfree() after calling tpinit(). The TPINIT typed buffer structure includes the following members:

```
char      usrname[MAXTIDENT+2];
char      cltname[MAXTIDENT+2];
char      passwd[MAXTIDENT+2];
char      grpname[MAXTIDENT+2];
long      flags;
long      datalen;
long      data;
```

The values of usrname, cltname, grpname, and passwd are all NULL-terminated strings. usrname is a name representing the caller. cltname is a client name whose semantics are application defined. The value sysclient is reserved by the system for the cltname field. The usrname and cltname fields are associated with the client at tpinit() time and are used for both broadcast notification and administrative statistics retrieval. They should not have more characters than MAXTIDENT, which is defined as 30. passwd is an application password in unencrypted format that is used for validation against the application password. The passwd is limited to 30 characters. grpname is used to associate the client with a resource manager group name. If grpname is set to a 0-length string, then the client is not associated with a resource manager and is in the default client group. The value of grpname must be the NULL string (0-length string) for Workstation clients. Note that grpname is not related to ACL GROUPS.

## Single-context Mode Versus Multicontext Mode

tpinit() has two modes of operation: single-context mode and multicontext mode. In single-context mode, a process may join at most one application at any one time. Multiple application threads may access this application. Single-context mode is specified by calling tpinit() with a NULL parameter or by calling it without specifying the TPMULTICONTEXTS flag in the flags field of the TPINIT structure. Single-context mode is also specified when tpinit() is called implicitly by another ATMI function. The context state for a process operating in single-context mode is TPSINGLECONTEXT.

**Note:** The TPMULTICONTEXTS mode of tpinit will continue to work properly when the TMNOTHREADS environment variable is set to "yes".

In single-context mode, if tpinit() is called more than once (that is, if it is called after the client has already joined the application), no action is taken and success is returned.

Multicontext mode is entered by calling tpinit() with the TPMULTICONTEXTS flag set in the flags field of the TPINIT structure. In multicontext mode, each call to tpinit() results in the creation of a separate application association.

An application association is a context that associates a process and a BEA Tuxedo ATMI application. A client may have associations with multiple BEA Tuxedo ATMI applications, and may also have multiple associations with the same application. All of a client's associations must be made to applications running the same release of the BEA Tuxedo ATMI system, and either all associations must be native clients or all associations must be Workstation clients.

For native clients, the value of the TUXCONFIG environment variable is used to identify the application to which the new association will be made. For Workstation clients, the value of the WSNADDR or WSENVFILE environment variable is used to identify the application to which the new association will be made. The context for the current thread is set to the new association.

In multicontext mode, the application can get a handle for the current context by calling tpgetctxt() and pass that handle as a parameter to tpsetctxt(), thus setting the context in which a particular thread or process will operate.

Mixing single-context mode and multicontext mode is not allowed. Once an application has chosen one of these modes, calling tpinit() in the other mode is not allowed unless tpterm() is first called for all application associations.

## TPINFO Structure Field Descriptions

In addition to controlling multicontext and single-context modes, the setting of flags is used to indicate both the client-specific notification mechanism and the mode of system access. These

two settings may override the application default. If these settings cannot override the application default, tpinit() prints a warning in a log file, ignores the setting, and restores the application default setting in the flags field upon return from tpinit(). For client notification, the possible values for flags are as follows:

TPU_SIG
: Select unsolicited notification by signals. This flag should be used only with single-threaded, single-contexted applications; it cannot be used when the TPMULTICONTEXTS flag is set.

TPU_DIP
: Select unsolicited notification by dip-in.

TPU_THREAD
: Select THREAD notification in a separate thread managed by the BEA Tuxedo ATMI system. This flag is allowed only on platforms that support multithreading. If TPU_THREAD is specified on a platform that does not support multithreading, it is considered an invalid argument and will result in an error return with tperrno set to TPEINVAL.

TPU_IGN
: Ignore unsolicited notification.

Only one of the above flags can be used at a time. If the client does not select a notification method via the flags field, then the application default method will be set in the flags field upon return from tpinit().

For setting the mode of system access, the possible values for flags are as follows:

TPSA_FASTPATH
: Set system access to fastpath.

TPSA_PROTECTED
: Set system access to protected.

Only one of the above flags can be used at a time. If the client does not select a notification method or a system access mode via the flags field, then the application default method(s) will be set in the flags field upon return from tpinit(). See UBBCONFIG(5) for details on both client notification methods and system access modes.

If your application uses multithreading and/or multicontexting, you must set the following flag:

TPMULTICONTEXTS
: See description in “Single-context Mode Versus Multicontext Mode.”

datalen is the length of the application-specific data that follows. The buffer type switch entry for the TPINIT typed buffer sets this field based on the total size passed in for the typed buffer

(the application data size is the total size less the size of the TPINIT structure itself plus the size of the data placeholder as defined in the structure). data is a place holder for variable length data that is forwarded to an application-defined authentication service. It is always the last element of this structure.

A macro, TPINITNEED, is available to determine the size TPINIT buffer necessary to accommodate a particular desired application specific data length. For example, if 8 bytes of application-specific data are desired, TPINITNEED(8) will return the required TPINIT buffer size.

A NULL value for tpinfo is allowed for applications not making use of the authentication feature of the BEA Tuxedo ATMI system. Clients using a NULL argument will get: defaults of 0-length strings for usrname, cltname and passwd; no flags set; and no application data.

## Return Values

Upon failure, tpinit() leaves the calling process in its original context, returns -1, and sets tperrno to indicate the error condition. Also, tpurcode() is set to the value returned by the AUTHSVR(5) server.

## Errors

Upon failure, tpinit() sets tperrno to one of the following values:

**[**TPEINVAL**]**
: Invalid arguments were specified. *tpinfo* is non-NULL and does not point to a typed buffer of type TPINIT.

**[**TPENOENT**]**
: The client cannot join the application because of space limitations.

**[**TPEPERM**]**
: The client cannot join the application because it does not have permission to do so or because it has not supplied the correct application password. Permission may be denied based on an invalid application password, failure to pass application-specific authentication, or use of restricted names. tpurcode() may be set by an application-specific authentication server to explain why the client cannot join the application.

**[**TPEPROTO**]**
: tpinit() has been called improperly. For example: (a) the caller is a server; (b) the TPMULTICONTEXTS flag has been specified in single-context mode; or (c) the TPMULTICONTEXTS flag has not been specified in multicontext mode.

**[**TPESYSTEM**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPEOS**]**
An operating system error has occurred.

## Interoperability

tpchkauth() and a non-NULL value for the TPINIT typed buffer argument of tpinit() are available only on sites running release 4.2 or later.

## Portability

The interfaces described in tpinit(3c) are supported on UNIX system, Windows, and MS-DOS operating systems. However, signal-based notification is not supported on 16-bit Windows or MS-DOS platforms. If it is selected at tpinit() time, then a userlog() message is generated and the method is automatically set to dip-in.

## Environment Variables

TUXCONFIG
Used within tpinit() when invoked by a native client. It indicates the application to which the client should connect. Note that this environment variable is referenced only when tpinit() is called. Subsequent calls make use of the application context.

WSENVFILE
Used within tpinit() when invoked by a Workstation client. It indicates a file containing environment variable settings that should be set in the caller's environment. See compilation(5) for details on environment variable settings necessary for Workstation clients. Note that this file is processed only when tpinit() is called and not before.

WSNADDR
Used within tpinit() when invoked by a Workstation client. It indicates the network addresses of the workstation listener that is to be contacted for access to the application. This variable is required for Workstation clients and is ignored for native clients.

TCP/IP addresses may be specified in the following forms:

*//host.name:port_number*

*//#.#.#.#:port_number*

In the first format, the domain finds an address for *hostname* using the local name resolution facilities (usually DNS). *hostname* must be the local machine, and the local

name resolution facilities must unambiguously resolve *hostname* to the address of the local machine.

In the second format, the string #.#.#.# is in dotted-decimal format. In dotted-decimal format, each # should be a number from 0 to 255. This dotted-decimal number represents the IP address of the local machine.

In both of the above formats, *port_number* is the TCP port number at which the domain process will listen for incoming requests. *port_number* can either be a number between 0 and 65535 or a name. If *port_number* is a name, then it must be found in the network services database on your local machine.

The address can also be specified in hexadecimal format when preceded by the characters 0x. Each character after the initial 0x is a number between 0 and 9 or a letter between A and F (case insensitive). The hexadecimal format is useful for arbitrary binary network addresses such as IPX/SPX or TCP/IP.

The address can also be specified as an arbitrary string. The value should be the same as that specified for the NLSADDR parameter in the NETWORK section of the configuration file.

More than one address can be specified if desired by specifying a comma-separated list of pathnames for WSNADDR. Addresses are tried in order until a connection is established. Any member of an address list can be specified as a parenthesized grouping of pipe-separated network addresses. For example:

```
WSNADDR=(//m1.acme.com:3050|//m2.acme.com:3050),//m3.acme.com:3050
```

For users running under Windows, the address string looks like the following:

```
set WSNADDR=(//m1.acme.com:3050^|//m2.acme.com:3050),//m3.acme.com:3050
```

Because the pipe symbol ( | ) is considered a special character in Windows, it must be preceded by a carat ( ^ )—an escape character in the Windows environment—when it is specified on the command line. However, if WSNADDR is defined in an envfile, the BEA Tuxedo ATMI system gets the values defined by WSNADDR through the tuxgetenv(3c) function. In this context, the pipe symbol ( | ) is not considered a special character, so you do not need to escape it with a carat (^).

The BEA Tuxedo ATMI system randomly selects one of the parenthesized addresses. This strategy distributes the load randomly across a set of listener processes. Addresses are tried in order until a connection is established. Use the value specified in the application configuration file for the workstation listener to be called. If the value begins with the characters 0x, it is interpreted as a string of hex-digits; otherwise, it is interpreted as ASCII characters.

WSFADDR

Used within tpinit() when invoked by a Workstation client. It specifies the network address used by the Workstation client when connecting to the workstation listener or workstation handler. This variable, along with the WSFRANGE variable, determines the range of TCP/IP ports to which a Workstation client will attempt to bind before making an outbound connection. This address must be a TCP/IP address. The port portion of the TCP/IP address represents the base address from which a range of TCP/IP ports can be bound by the Workstation client. The WSFRANGE variable specifies the size of the range. For example, if this address is //mymachine.bea.com:30000 and WSFRANGE is 200, then all native processes attempting to make outbound connections from this *LMID* will bind a port on mymachine.bea.com between 30000 and 30200. If not set, this variable defaults to the empty string, which implies the operating system chooses a local port randomly.

WSFRANGE

Used within tpinit() when invoked by a Workstation client. It specifies the range of TCP/IP ports to which a Workstation client process will attempt to bind before making an outbound connection. The WSFADDR parameter specifies the base address of the range. For example, if the WSFADDR parameter is set to //mymachine.bea.com:30000 and WSFRANGE is set to 200, then all native processes attempting to make outbound connections from this *LMID* will bind a port on mymachine.bea.com between 30000 and 30200. The valid range is 1-65535. The default is 1.

WSDEVICE

Used within tpinit() when invoked by a Workstation client. It indicates the device name to be used to access the network. This variable is used by Workstation clients and ignored for native clients. Note that certain supported transport level network interfaces do not require a device name; for example, sockets and NetBIOS. Workstation clients supported by such interfaces need not specify WSDEVICE.

WSTYPE

Used within tpinit() when invoked by a Workstation client to negotiate encode/decode responsibilities with the native site. This variable is optional for Workstation clients and ignored for native clients.

WSRPLYMAX

Used by tpinit() to set the maximum amount of core memory that should be used for buffering application replies before they are dumped to file. The default for this parameter 256,000 bytes. For more information, see the programming documentation for your instantiation.

TMMINENCRYPTBITS

Used to establish the minimum level of encryption required to connect to the BEA Tuxedo ATMI system. “0” means no encryption, while “56” and “128” specify the encryption key length (in bits). The link-level encryption value of 40 bits is also provided for backward

compatibility. If this minimum level of encryption cannot be met, link establishment will fail. The default is “0”.

TMMAXENCRYPTBITS
: Used to negotiate the level of encryption up to this level when connecting to the BEA Tuxedo ATMI system. “0” means no encryption, while “56” and “128” specify the encryption length (in bits). The link-level encryption value of 40 bits is also provided for backward compatibility. The default is “128.”

### Warning

Signal-based notification is not allowed in multicontext mode. In addition, signal restrictions may prevent the system from using signal-based notification even though it has been selected by a client. When this happens, the system generates a log message that it is switching notification for the selected client to dip-in and the client is notified then and thereafter via dip-in notification. (See the description of the NOTIFY parameter in the RESOURCES section of UBBCONFIG(5) for a detailed discussion of notification methods.)

Because signaling of clients is always done by the system, the behavior of notification is always consistent, regardless of where the originating notification call is made. Therefore to use signal-based notification:

- A native client must be running as an application administrator.
- A Workstation client is not required to be running as the application administrator

The ID for the application administrator is identified as part of the configuration for the application.

If signal-based notification is selected for a client, then certain ATMI calls may fail, returning TPGOTSIG due to receipt of an unsolicited message if TPSIGRSTRT is not specified.

### See Also

Introduction to the C Language Application-to-Transaction Monitor Interface, tpgetctxt(3c), tpsetctxt(3c), tpterm(3c)

## tpkey_close(3c)

### Name

tpkey_close()—Closes a previously opened key handle.

## Synopsis

```
#include <atmi.h>
int tpkey_close(TPKEY hKey, long flags)
```

## Description

`tpkey_close()` releases a previously opened key handle and all resources associated with it. Any sensitive information, such as the principal's private key, is erased from memory.

Key handles can be opened in one of two ways:

- By an explicit call to `tpkey_open()`
- As output from `tpenvelope()`

It is the application's responsibility to release key resources by calling `tpkey_close()`. Once a process closes a key, the process can no longer use the key handle to register a message buffer for digital signature or encryption. If the process opened the key using `tpkey_open()` with the `TPKEY_AUTOSIGN` or `TPKEY_AUTOENCRYPT` flag specified, the key handle no longer applies to future communication operations after the key is closed.

Even though a key is closed, however, the key handle continues to be valid for any associated signature or encryption request registered before the key was closed. When the last buffer associated with a closed key is freed or overwritten, resources attributable to the key are released.

The *`flags`* argument is reserved for future use and must be set to 0.

## Return Values

On failure, this function returns -1 and sets `tperrno` to indicate the error condition.

## Errors

**[**`TPEINVAL`**]**
Invalid arguments were given. For example, the value of *`hKey`* is not a valid key.

**[**`TPESYSTEM`**]**
An error occurred. Consult the system error log file for details.

## See Also

`tpenvelope(3c)`, `tpkey_getinfo(3c)`, `tpkey_open(3c)`, `tpkey_setinfo(3c)`

## tpkey_getinfo(3c)

### Name

`tpkey_getinfo()`—Gets information associated with a key handle.

### Synopsis

```
#include <atmi.h>
int tpkey_getinfo(TPKEY hKey, char *attribute_name, void *value, long
*value_len, long flags)
```

### Description

`tpkey_getinfo()` reports information about a key handle. A key handle represents a specific principal's key and the information associated with it.

The key under examination is identified by the *`hKey`* input parameter. The attribute for which information is desired is identified by the *`attribute_name`* input parameter. Some attributes are specific to a cryptographic service provider, but the following core set of attributes should be supported by all providers.

| Attribute | Value |
|---|---|
| `PRINCIPAL` | The name identifying the principal associated with the key (key handle), represented as a NULL-terminated character string. |
| `PKENCRYPT_ALG` | An ASN.1 Distinguished Encoding Rules (DER) *object identifier* of the public key algorithm used by the key for public key encryption.<br>The object identifier for RSA is identified in the following table. |
| `PKENCRYPT_BITS` | The key length of the public key algorithm (RSA modulus size). The value must be within the range of 512 to 2048 bits, inclusive. |
| `SIGNATURE_ALG` | An ASN.1 DER *object identifier* of the digital signature algorithm used by the key for digital signature.<br>The object identifiers for RSA and DSA are identified in the following table. |
| `SIGNATURE_BITS` | The key length of the digital signature algorithm (RSA modulus size). The value must be within the range of 512 to 2048 bits, inclusive. |

| Attribute | Value |
|---|---|
| ENCRYPT_ALG | An ASN.1 DER *object identifier* of the symmetric key algorithm used by the key for bulk data encryption.<br>The object identifiers for DES, 3DES, and RC2 are identified in the following table. |
| ENCRYPT_BITS | The key length of the symmetric key algorithm. The value must be within the range of 40 to 128 bits, inclusive.<br>When an algorithm with a fixed key length is set in ENCRYPT_ALG, the ENCRYPT_BITS value is automatically set to the fixed key length. For example, if ENCRYPT_ALG is set to DES, the ENCRYPT_BITS value is automatically set to 56. |
| DIGEST_ALG | An ASN.1 DER *object identifier* of the message digest algorithm used by the key for digital signature.<br>The object identifiers for MD5 and SHA-1 are identified in the following table. |
| PROVIDER | The name of the cryptographic service provider. |
| VERSION | The version number of the cryptographic service provider's software. |

The ASN.1 DER algorithm object identifiers supported by the default public key implementation are given in the following table.

| ASN.1 DER Algorithm Object Identifier | Algorithm |
|---|---|
| { 0x06, 0x08, 0x2a, 0x86, 0x48, 0x86, 0xf7, 0x0d, 0x02, 0x05 } | MD5 |
| { 0x06, 0x05, 0x2b, 0x0e, 0x03, 0x02, 0x1a } | SHA1 |
| { 0x06, 0x09, 0x2a, 0x86, 0x48, 0x86, 0xf7, 0x0d, 0x01, 0x01, 0x01 } | RSA |
| { 0x06, 0x05, 0x2b, 0x0e, 0x03, 0x02, 0x0c } | DSA |
| { 0x06, 0x05, 0x2b, 0x0e, 0x03, 0x02, 0x07 } | DES |
| { 0x06, 0x08, 0x2a, 0x86, 0x48, 0x86, 0xf7, 0x0d, 0x03, 0x07 } | 3DES |
| { 0x06, 0x08, 0x2a, 0x86, 0x48, 0x86, 0xf7, 0x0d, 0x03, 0x02 } | RC2 |

The information associated with the specified *attribute_name* parameter will be stored in the memory location indicated by *value*. The maximum amount of data that can be stored at this location is specified by the caller in *value_len*.

After tpkey_getinfo() completes, *value_len* is set to the size of the data actually returned (including a terminating NULL value for string values). If the number of bytes that need to be returned exceeds *value_len*, tpkey_getinfo() fails (with the TPELIMIT error code) and sets *value_len* to the required amount of space.

The *flags* argument is reserved for future use and must be set to 0.

### Return Values

On failure, this function returns -1 and sets tperrno to indicate the error condition.

### Errors

**[**TPEINVAL**]**
Invalid arguments were given. For example, *hKey* is not a valid key.

**[**TPESYSTEM**]**
An error occurred. Consult the system error log file for details.

**[**TPELIMIT**]**
Insufficient space was provided to hold the requested attribute value.

**[**TPENOENT**]**
The requested attribute is not associated with this key.

### See Also

tpkey_close(3c), tpkey_open(3c), tpkey_setinfo(3c)

## tpkey_open(3c)

### Name

tpkey_open()—Opens a key handle for digital signature generation, message encryption, or message decryption.

### Synopsis

```
#include <atmi.h>
int tpkey_open(TPKEY *hKey, char *principal_name, char *location, char
*identity_proof, long proof_len, long flags)
```

## Description

`tpkey_open()` makes a key handle available to the calling process. A key handle represents a specific principal's key and the information associated with it.

A key may be used for one or more of the following purposes:

- Generating a digital signature, which protects a typed message buffer's content and proves that a specific principal originated the message. (A principal may be a person or a process.) This type of key is a private key and is available only to the key's owner.

  Calling `tpkey_open()` with the principal's name and either the `TPKEY_SIGNATURE` or `TPKEY_AUTOSIGN` flag returns a handle to the principal's private key and digital certificate.

- Verifying a digital signature, which proves that a typed message buffer's content remains unaltered and that a specific principal originated the message.

  Signature verification does not require a call to `tpkey_open()`; the verifying process uses the public key specified in the digital certificate accompanying the digitally signed message to verify the signature.

- Encrypting a message buffer destined for a specific principal. This type of key is available to any process with access to the principal's public key and digital certificate.

  Calling `tpkey_open()` with the principal's name and either the `TPKEY_ENCRYPT` or `TPKEY_AUTOENCRYPT` flag returns a handle to the principal's public key via the principal's digital certificate.

- Decrypting a message buffer intended for a specific principal. This type of key is a private key and is available only to the key's owner.

  Calling `tpkey_open()` with the principal's name and the `TPKEY_DECRYPT` flag returns a handle to the principal's private key and digital certificate.

The key handle returned by `tpkey_open()` is stored in *`*hKey`*, the value of which cannot be NULL.

The *`principal_name`* input parameter specifies the key owner's identity. If the value of *`principal_name`* is a NULL pointer or an empty string, a default identity is assumed. The default identity may be based on the current login session, the current operating system account, or another attribute such as a local hardware device.

The file location of a key may be passed into the *`location`* parameter. If the underlying key management provider does not require a location parameter, the value of this parameter may be NULL.

To authenticate the identity of *principal_name*, proof material such as a password or pass phrase may be required. If required, the proof material should be referenced by *identity_proof*. Otherwise, the value of this parameter may be NULL.

The length of the proof material (in bytes) is specified by *proof_len*. If *proof_len* is 0, *identity_proof* is assumed to be a NULL-terminated character string.

The type of key access required for a key's mode of operation is specified by the *flags* parameter:

TPKEY_SIGNATURE:
: This private key is available to generate digital signatures.

TPKEY_AUTOSIGN:
: Whenever this process transmits a message buffer, the public key software uses the signer's private key to generate a digital signature and then attaches the digital signature to the buffer. TPKEY_SIGNATURE is implied.

TPKEY_ENCRYPT:
: This public key is available to identify the recipient of an encrypted message.

TPKEY_AUTOENCRYPT:
: Whenever this process transmits a message buffer, the public key software encrypts the message content, uses the recipient's public key to generate an encryption envelope, and then attaches the encryption envelope to the buffer. TPKEY_ENCRYPT is implied.

TPKEY_DECRYPT:
: This private key is available for decryption.

Any combination of one or more of these flag values is allowed. If a key is used only for encryption (TPKEY_ENCRYPT), *identity_proof* is not required and may be set to NULL.

## Return Values

Upon successful completion, **hKey* is set to a value that represents this key, for use by other functions such as `tpsign()` and `tpseal()`.

On failure, this function returns -1 and sets `tperrno` to indicate the error condition.

## Errors

[TPEINVAL]
: Invalid arguments were given. For example, the value of *hKey* is NULL or the *flags* parameter is not set correctly.

**[**TPEPERM**]**
Permission failure. The cryptographic service provider was not able to access a private key for this principal, given the proof information and current environment.

**[**TPESYSTEM**]**
A system error occurred. Consult the systems error log file for details.

### See Also

tpkey_close(3c), tpkey_getinfo(3c), tpkey_setinfo(3c)

## tpkey_setinfo(3c)

### Name

tpkey_setinfo()—Sets optional attribute parameters associated with a key handle.

### Synopsis

```
#include <atmi.h>
int tpkey_setinfo(TPKEY hKey, char *attribute_name, void *value, long
value_len, long flags)
```

### Description

tpkey_setinfo() sets an optional attribute parameter for a key handle. A key handle represents a specific principal's key and the information associated with it.

The key for which information is to be modified is identified by the *hKey* input parameter. The attribute for which information is to be modified is identified by the *attribute_name* input parameter. Some attributes may be specific to a certain cryptographic service provider, but the core set of attributes presented on the tpkey_getinfo(3c) reference page should be supported by all providers.

The information to be associated with the *attribute_name* parameter is stored in the memory location indicated by *value*. If the data content of *value* is self-describing, *value_len* is ignored (and may be 0). Otherwise, *value_len* must contain the length of data in *value*.

The *flags* argument is reserved for future use and must be set to 0.

### Return Values

On failure, this function returns -1 and sets tperrno to indicate the error condition.

### Errors

**[**TPEINVAL**]**
Invalid arguments were given. For example, *hKey* is not a valid key or *attribute_name* refers to a read-only value.

**[**TPELIMIT**]**
The *value* provided is too large.

**[**TPESYSTEM**]**
An error occurred. Consult the system error log file for details.

**[**TPENOENT**]**
The requested attribute is not recognized by the key's cryptographic service provider.

### See Also

tpkey_close(3c), tpkey_getinfo(3c), tpkey_open(3c)

## tpnotify(3c)

### Name

tpnotify()—Routine for sending notification by client identifier.

### Synopsis

```
#include <atmi.h>
int tpnotify(CLIENTID *clientid, char *data, long len, long flags)
```

### Description

tpnotify() allows a client or server to send an unsolicited message to an individual client.

*clientid* is a pointer to a client identifier saved from the TPSVCINFO structure of a previous or current service invocation, or passed to a client via some other communications mechanism (for example, retrieved via the administration interface).

The data portion of the request is pointed to by *data*, a buffer previously allocated by tpalloc(). *len* specifies how much of *data* to send. Note that if *data* points to a buffer type that does not require a length to be specified, (for example, an FML fielded buffer) then *len* is ignored (and may be 0). Also, *data* may be NULL in which case *len* is ignored.

Upon successful return from tpnotify(), the message has been delivered to the system for forwarding to the identified client. If the TPACK flag was set, a successful return means the

message has been received by the client. Furthermore, if the client has registered an unsolicited message handler, the handler will have been called.

The following is a list of valid *flags*:

TPACK
: The request is sent and the caller blocks until an acknowledgement message is received from the target client.

TPNOBLOCK
: The request is not sent if a blocking condition exists in sending the notification (for example, the internal buffers into which the message is transferred are full).

TPNOTIME
: This flag signifies that the caller is willing to block indefinitely and wants to be immune to blocking timeouts. Transaction timeouts may still occur.

TPSIGRSTRT
: If a signal interrupts any underlying system calls, then the interrupted system call is reissued.

  Unless the TPACK flag is set, tpnotify() does not wait for the message to be delivered to the client.

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tpnotify().

## Return Values

Upon failure, tpnotify() returns -1 and sets tperrno to indicate the error condition. If a call fails with a particular tperrno value, a subsequent call to tperrordetail(), with no intermediate ATMI calls, may provide more detailed information about the generated error. Refer to the tperrordetail(3c) reference page for more information.

## Errors

Upon failure, tpnotify() sets tperrno to one of the following values:

**[**TPEINVAL**]**
: Invalid arguments were given (for example, invalid flags).

**[**TPENOENT**]**
: The target client does not exist or does not have an unsolicited handler set and the TPACK flag is set.

[TPETIME]
A blocking timeout occurred and neither TPNOBLOCK nor TPNOTIME were specified, or TPACK was set but no acknowledgment was received and TPNOTIME was not specified. (A blocking timeout cannot occur if TPNOBLOCK and/or TPNOTIME is specified.)

[TPEBLOCK]
A blocking condition was found on the call and TPNOBLOCK was specified.

[TPGOTSIG]
A signal was received and TPSIGRSTRT was not specified.

[TPEPROTO]
tpnotify() was called improperly.

[TPESYSTEM]
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

[TPEOS]
An operating system error has occurred.

[TPERELEASE]
When the TPACK is set and the target is a client from a prior release of BEA Tuxedo that does not support the acknowledgment protocol.

### See Also

Introduction to the C Language Application-to-Transaction Monitor Interface, tpalloc(3c), tpbroadcast(3c), tpchkunsol(3c), tperrordetail(3c), tpinit(3c), tpsetunsol(3c), tpstrerrordetail(3c), tpterm(3c)

## tpopen(3c)

### Name

tpopen()—Routine for opening a resource manager.

### Synopsis

```
#include <atmi.h>
int tpopen(void)
```

## Description

`tpopen()` opens the resource manager to which the caller is linked. At most one resource manager can be linked to the caller. This function is used in place of resource manager-specific `open()` calls and allows a service routine to be free of calls that may hinder portability. Since resource managers differ in their initialization semantics, the specific information needed to open a particular resource manager is placed in a configuration file.

If a resource manager is already open (that is, `tpopen()` is called more than once), no action is taken and success is returned.

In a multithreaded application, a thread in the `TPINVALIDCONTEXT` state is not allowed to issue a call to `tpopen()`.

## Return Values

Upon failure, `tpopen()` returns -1 and sets `tperrno` to indicate the error condition.

## Errors

Upon failure, `tpopen()` sets `tperrno` to one of the following values:

**[**`TPERMERR`**]**
A resource manager failed to open correctly. More information concerning the reason a resource manager failed to open can be obtained by interrogating a resource manager in its own specific manner. Note that any calls to determine the exact nature of the error hinder portability.

**[**`TPEPROTO`**]**
`tpopen()` was called in an improper context (for example, by a client that has not joined a BEA Tuxedo system server group).

**[**`TPESYSTEM`**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**`TPEOS`**]**
An operating system error has occurred.

## See Also

`tpclose(3c)`

# tppost(3c)

## Name

tppost()—Posts an event.

## Synopsis

```
#include <atmi.h>
int tppost(char *eventname, char *data, long len, long flags)
```

## Description

The caller uses tppost() to post an event and any accompanying data. The event is named by *eventname* and *data*, if not NULL, points to the data. The posted event and its data are dispatched by the BEA Tuxedo ATMI EventBroker to all subscribers whose subscriptions successfully evaluate against *eventname* and whose optional filter rules successfully evaluate against *data*.

*eventname* is a NULL-terminated string of at most 31 characters. *eventname*'s first character cannot be a dot (".") as this character is reserved as the starting character for all events defined by the BEA Tuxedo ATMI system itself.

If *data* is non-NULL, it must point to a buffer previously allocated by tpalloc() and *len* should specify the amount of data in the buffer that should be posted with the event. Note that if *data* points to a buffer of a type that does not require a length to be specified (for example, an FML fielded buffer), then *len* is ignored. If *data* is NULL, *len* is ignored and the event is posted with no data.

When tppost() is used within a transaction, the transaction boundary can be extended to include those servers and/or stable-storage message queues notified by the EventBroker. When a transactional posting is made, some of the recipients of the event posting are notified on behalf of the poster's transaction (for example, servers and queues), while some are not (for example, clients).

If the poster is within a transaction and the TPNOTRAN flag is not set, the posted event goes to the EventBroker in transaction mode such that it dispatches the event as part of the poster's transaction. The broker dispatches transactional event notifications only to those service routine and stable-storage queue subscriptions that used the TPEVTRAN bit setting in the *ctl*→*flags* parameter passed to tpsubscribe(). Client notifications, and those service routine and stable-storage queue subscriptions that did not use the TPEVTRAN bit setting in the *ctl*→*flags* parameter passed to tpsubscribe(), are also dispatched by the EventBroker but not as part of the posting process's transaction.

If the poster is outside a transaction, tppost() is a one-way post with no acknowledgement when the service associated with the event fails. This occurs even when TPEVTRAN is set for that event (using the *ctl→flags* parameter passed to tpsubscribe()). If the poster is in a transaction, then tppost() returns TPESVCFAIL when the associated service fails in the event.

The following is a list of valid *flags*:

TPNOTRAN
: If the caller is in transaction mode and this flag is set, then the event posting is not made on behalf of the caller's transaction. A caller in transaction mode that sets this flag is still subject to the transaction timeout (and no other) when posting events. If the event posting fails, the caller's transaction is not affected.

TPNOREPLY
: Informs tppost() not to wait for the EventBroker to process all subscriptions for *eventname* before returning. When TPNOREPLY is set, tpurcode() is set to zero regardless of whether tppost() returns successfully or not. When the caller is in transaction mode, this setting cannot be used unless TPNOTRAN is also set.

TPNOBLOCK
: The event is not posted if a blocking condition exists. If such a condition occurs, the call fails and tperrno is set to TPEBLOCK. When TPNOBLOCK is not specified and a blocking condition exists, the caller blocks until the condition subsides or a timeout occurs (either transaction or blocking timeout).

TPNOTIME
: This flag signifies that the caller is willing to block indefinitely and wants to be immune to blocking timeouts. Transaction timeouts may still occur.

TPSIGRSTRT
: If a signal interrupts any underlying system calls, then the interrupted system call is reissued. When TPSIGRSTRT is not specified and a signal interrupts a system call, then tppost() fails and tperrno is set to TPGOTSIG.

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tppost().

## Return Values

Upon successful return from tppost(), tpurcode() contains the number of event notifications dispatched by the EventBroker on behalf of *eventname* (that is, postings for those subscriptions whose event expression evaluated successfully against *eventname* and whose filter rule evaluated successfully against *data*). Upon return where tperrno is set to TPESVCFAIL,

tpurcode() contains the number of non-transactional event notifications dispatched by the EventBroker on behalf of *eventname*.

Upon failure, tppost() returns -1 sets tperrno to indicate the error condition.

## Errors

Upon failure, tppost() sets tperrno to one of the following values. (Unless otherwise noted, failure does not affect the caller's transaction, if one exists.)

**[**TPEINVAL**]**
Invalid arguments were given (for example, *eventname* is NULL).

**[**TPENOENT**]**
Cannot access the BEA Tuxedo User EventBroker.

**[**TPETRAN**]**
The caller is in transaction mode, TPNOTRAN was not set and tppost() contacted an EventBroker that does not support transaction propagation (that is, TMUSREVT(5) is not running in a BEA Tuxedo ATMI system group that supports transactions).

**[**TPETIME**]**
This error code indicates that either a timeout has occurred or tppost() has been attempted, in spite of the fact that the current transaction is already marked rollback only.

If the caller is in transaction mode, then either the transaction is already rollback only or a transaction timeout has occurred. The transaction is marked abort-only. If the caller is not in transaction mode, a blocking timeout has occurred. (A blocking timeout cannot occur if TPNOBLOCK and/or TPNOTIME is specified.)

If a transaction timeout has occurred, then, with one exception, any attempts to send new requests or receive outstanding replies will fail with TPETIME until the transaction has been aborted. The exception is a request that does not block, expects no reply, and is not sent on behalf of the caller's transaction (that is, tpacall() with TPNOTRAN, TPNOBLOCK, and TPNOREPLY set).

When tppost() fails inside a transaction, the transaction is put into the TX_ROLLBACK_ONLY state. This state is treated, for most purposes, as though it were equivalent to a timeout. All further ATMI calls for this transaction (with the exception of those issued in the circumstances described in the previous paragraph) will fail with TPETIME.

**[**TPESVCFAIL**]**
The EventBroker encountered an error posting a transactional event to either a service routine or to a stable storage queue on behalf of the caller's transaction. The caller's current transaction is marked abort-only. When this error is returned, tpurcode() contains the number of non-transactional event notifications dispatched by the EventBroker on behalf of *eventname*; transactional postings are not counted since their effects will be aborted upon completion of the transaction. Note that so long as the transaction has not timed out, further communication may be performed before aborting the transaction and that any work performed on behalf of the caller's transaction will be aborted upon transaction completion (that is, for subsequent communication to have any lasting effect, it should be done with TPNOTRAN set).

**[**TPEBLOCK**]**
A blocking condition exists and TPNOBLOCK was specified.

**[**TPGOTSIG**]**
A signal was received and TPSIGRSTRT was not specified.

**[**TPEPROTO**]**
tppost() was called improperly.

**[**TPESYSTEM**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPEOS**]**
An operating system error has occurred.

### See Also

tpsubscribe(3c), tpunsubscribe(3c), EVENTS(5), TMSYSEVT(5), TMUSREVT(5)

## tprealloc(3c)

### Name

tprealloc()—Routine to change the size of a typed buffer.

### Synopsis

```
#include <atmi.h>
char * tprealloc(char *ptr, long size)
```

## Description

tprealloc() changes the size of the buffer pointed to by *ptr* to *size* bytes and returns a pointer to the new (possibly moved) buffer. Similar to tpalloc(), the size of the buffer will be at least as large as the larger of *size* and dfltsize, where dfltsize is the default buffer size specified in tmtype_sw. If the larger of the two is less than or equal to zero, then the buffer is unchanged and NULL is returned. A buffer's type remains the same after it is reallocated. After this function returns successfully, the returned pointer should be used to reference the buffer; *ptr* should no longer be used. The buffer's contents will not change up to the lesser of the new and old sizes.

Some buffer types require initialization before they can be used. tprealloc() reinitializes a buffer (in a communication manager-specific manner) after it is reallocated and before it is returned. Thus, the buffer returned to the caller is ready for use.

A thread in a multithreaded application may issue a call to tprealloc() while running in any context state, including TPINVALIDCONTEXT.

## Return Values

Upon successful completion, tprealloc() returns a pointer to a buffer of the appropriate type aligned on a long word.

Upon failure, tprealloc() returns NULL and sets tperrno to indicate the error condition.

## Errors

If the reinitialization function fails, tprealloc() fails, returning NULL and the contents of the buffer pointed to by *ptr* may not be valid. Upon failure, tprealloc() sets tperrno to one of the following values:

**[**TPEINVAL**]**
: Invalid arguments were given (for example, *ptr* does not point to a buffer originally allocated by tpalloc()).

**[**TPEPROTO**]**
: tprealloc() was called improperly.

**[**TPESYSTEM**]**
: A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPEOS**]**
: An operating system error has occurred.

### Usage

If buffer reinitialization fails, `tprealloc()` fails returning NULL and the contents of the buffer pointed to by *`ptr`* may not be valid. This function should not be used in concert with `malloc()`, `realloc()` or `free()` in the C library (for example, a buffer allocated with `tprealloc()` should not be freed with `free()`).

### See Also

`tpalloc(3c)`, `tpfree(3c)`, `tptypes(3c)`

## tprecv(3c)

### Name

`tprecv()`—Routine for receiving a message in a conversational connection.

### Synopsis

```
#include <atmi.h>
int tprecv(int cd, char **data, long *len, long flags, long \
   *revent)
```

### Description

`tprecv()` is used to receive data sent across an open connection from another program. `tprecv()`'s first argument, *`cd`*, specifies on which open connection to receive data. *`cd`* is a descriptor returned from either `tpconnect()` or the `TPSVCINFO` parameter to the service. The second argument, *`data`*, is the address of a pointer to a buffer previously allocated by `tpalloc()`.

*`data`* must be the address of a pointer to a buffer previously allocated by `tpalloc()` and *`len`* should point to a long that `tprecv()` sets to the amount of data successfully received. Upon successful return, *`*data`* points to a buffer containing the reply and *`*len`* contains the size of the buffer. `FML` and `FML32` buffers often assume a minimum size of 4096 bytes; if the reply is larger than 4096 bytes, the size of the buffer is increased to a size large enough to accommodate the data being returned.

Buffers on the sending side that may be only partially filled (for example, FML or STRING buffers) will have only the amount that is used sent. The system may then enlarge the received data size by some arbitrary amount. This means that the receiver may receive a buffer that is smaller than what was originally allocated by the sender, yet larger than the data that was sent.

The receive buffer may grow, or it may shrink, and its address almost invariably changes, as the system swaps buffers around internally. To determine whether (and how much) a reply buffer

changed in size, compare its total size before tprecv() was issued with *len*. See "Introduction to the C Language Application-to-Transaction Monitor Interface" for more information about buffer management.

If *len* is 0, then no data was received and neither *data* nor the buffer it points to were modified. It is an error for *data*, *data* or *len* to be NULL.

tprecv() can be issued only by the program that does not have control of the connection.

The following is a list of valid *flags*:

TPNOCHANGE
: By default, if a buffer is received that differs in type from the buffer pointed to by **data*, then **data*'s buffer type changes to the received buffer's type so long as the receiver recognizes the incoming buffer type. When this flag is set, the type of the buffer pointed to by **data* is not allowed to change. That is, the type and subtype of the received buffer must match the type and subtype of the buffer pointed to by **data*.

TPNOBLOCK
: tprecv() does not wait for data to arrive. If data is already available to receive, then tprecv() gets the data and returns. When this flag is not specified and no data is available to receive, the caller blocks until data arrives.

TPNOTIME
: This flag signifies that the caller is willing to block indefinitely and wants to be immune to blocking timeouts. Transaction timeouts will still affect the program.

TPSIGRSTRT
: If a signal interrupts the underlying receive system call, then the call is reissued.

If an event exists for the descriptor, *cd*, then tprecv() will return setting tperrno to TPEEVENT. The event type is returned in *revent*. Data can be received along with the TPEV_SVCSUCC, TPEV_SVCFAIL, and TPEV_SENDONLY events. Valid events for tprecv() are as follows:

TPEV_DISCONIMM
: Received by the subordinate of a conversation, this event indicates that the originator of the conversation has either issued an immediate disconnect on the connection via tpdiscon(), or it issued tpreturn(), tpcommit() or tpabort() with the connection still open. This event is also returned to the originator or subordinate when a connection is broken due to a communications error (for example, a server, machine, or network failure). Because this is an immediate disconnection notification (that is, abortive rather than orderly), data in transit may be lost. If the two programs were participating in the same transaction, then the transaction is marked abort-only. The descriptor used for the connection is no longer valid.

TPEV_SENDONLY
: The program on the other end of the connection has relinquished control of the connection. The recipient of this event is allowed to send data but cannot receive any data until it relinquishes control.

TPEV_SVCERR
: Received by the originator of a conversation, this event indicates that the subordinate of the conversation has issued `tpreturn()`. `tpreturn()` encountered an error that precluded the service from returning successfully. For example, bad arguments may have been passed to `tpreturn()` or `tpreturn()` may have been called while the service had open connections to other subordinates. Due to the nature of this event, any application defined data or return code are not available. The connection has been torn down and is no longer a valid descriptor. If this event occurred as part of the *cd* recipient's transaction, then the transaction is marked abort-only.

TPEV_SVCFAIL
: Received by the originator of a conversation, this event indicates that the subordinate service on the other end of the conversation has finished unsuccessfully as defined by the application (that is, it called `tpreturn()` with `TPFAIL` or `TPEXIT`). If the subordinate service was in control of this connection when `tpreturn()` was called, then it can pass an application defined return value and a typed buffer back to the originator of the connection. As part of ending the service routine, the server has torn down the connection. Thus, *cd* is no longer a valid descriptor. If this event occurred as part of the recipient's transaction, then the transaction is marked abort-only.

TPEV_SVCSUCC
: Received by the originator of a conversation, this event indicates that the subordinate service on the other end of the conversation has finished successfully as defined by the application (that is, it called `tpreturn()` with `TPSUCCESS`). As part of ending the service routine, the server has torn down the connection. Thus, *cd* is no longer a valid descriptor. If the recipient is in transaction mode, then it can either commit (if it is also the initiator) or abort the transaction causing the work done by the server (if also in transaction mode) to either commit or abort.

In a multithreaded application, a thread in the `TPINVALIDCONTEXT` state is not allowed to issue a call to `tprecv()`.

## Return Values

Upon return from `tprecv()` where *revent* is set to either `TPEV_SVCSUCC` or `TPEV_SVCFAIL`, the `tpurcode` global contains an application defined value that was sent as part of `tpreturn()`.

Upon failure, `tprecv()` returns -1 and sets `tperrno` to indicate the error condition. If a call fails with a particular `tperrno` value, a subsequent call to `tperrordetail()`, with no intermediate

ATMI calls, may provide more detailed information about the generated error. Refer to the `tperrordetail(3c)` reference page for more information.

## Errors

Upon failure, `tprecv()` sets `tperrno` to one of the following values:

**[**`TPEINVAL`**]**
Invalid arguments were given (for example, data is not the address of a pointer to a buffer allocated by `tpalloc()` or *`flags`* are invalid).

**[**`TPEOTYPE`**]**
Either the type and subtype of the incoming buffer are not known to the caller, or `TPNOCHANGE` was set in *`flags`* and the type and subtype of *`*data`* do not match the type and subtype of the incoming buffer. Regardless, neither *`*data`*, its contents nor *`*len`* are changed. If the conversation is part of the caller's current transaction, then the transaction is marked abort-only because the incoming buffer is discarded.

**[**`TPEBADDESC`**]**
*`cd`* is invalid.

**[**`TPETIME`**]**
This error code indicates that either a timeout has occurred or `tprecv()` has been attempted, in spite of the fact that the current transaction is already marked rollback only.

If the caller is in transaction mode, then either the transaction is already rollback only or a transaction timeout has occurred. The transaction is marked abort-only. If the caller is not in transaction mode, a blocking timeout has occurred. (A blocking timeout cannot occur if `TPNOBLOCK` and/or `TPNOTIME` is specified.) In either case, no changes are made to `*data` or its contents.

If a transaction timeout has occurred, then, with one exception, any attempts to send new requests or receive outstanding replies will fail with `TPETIME` until the transaction has been aborted. The exception is a request that does not block, expects no reply, and is not sent on behalf of the caller's transaction (that is, `tpacall()` with `TPNOTRAN`, `TPNOBLOCK`, and `TPNOREPLY` set).

When an ATMI call fails inside a transaction, the transaction is put into the `TX_ROLLBACK_ONLY` state. This state is treated, for most purposes, as though it were equivalent to a timeout. All further ATMI calls for this transaction (with the exception of those issued in the circumstances described in the previous paragraph) will fail with `TPETIME`.

**[**TPEEVENT**]**
An event occurred and its type is available in revent. There is a relationship between the [TPETIME] and the [TPEEVENT] return codes. While in transaction mode, if the receiving side of a conversation is blocked on tprecv and the sending side calls tpabort(), then the receiving side gets a return code of [TPEVENT] with an event of TPEV_DISCONIMM. If, however, the sending side calls tpabort() before the receiving side calls tprecv(), then the transaction may have already been removed from the GTT, which causes tprecv() to fail with the [TPETIME] code.

**[**TPEBLOCK**]**
A blocking condition exists and TPNOBLOCK was specified.

**[**TPGOTSIG**]**
A signal was received and TPSIGRSTRT was not specified.

**[**TPEPROTO**]**
tprecv() was called in an improper context (for example, the connection was established such that the calling program can only send data).

**[**TPESYSTEM**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPEOS**]**
An operating system error has occurred.

### Usage

A server can pass an application defined return value and typed buffer when calling tpreturn(). The return value is available in the global variable *tpurcode* and the buffer is available in *data*.

### See Also

tpalloc(3c), tpconnect(3c), tpdiscon(3c), tperrordetail(3c), tpsend(3c), tpservice(3c), tpstrerrordetail(3c)

## tpresume(3c)

### Name

tpresume()—Resumes a global transaction.

## Synopsis

```
#include <atmi.h>
int tpresume(TPTRANID *tranid, long flags)
```

## Description

`tpresume()` is used to resume work on behalf of a previously suspended transaction. Once the caller resumes work on a transaction, it must either suspend it with `tpsuspend()`, or complete it with one of `tpcommit()` or `tpabort()` at a later time.

The caller must ensure that its linked resource managers have been opened (via `tpopen()`) before it can resume work on any transaction.

`tpresume()` places the caller in transaction mode on behalf of the global transaction identifier pointed to by *`tranid`*. It is an error for *`tranid`* to be NULL.

Currently, *`flags`* are reserved for future use and must be set to 0.

In a multithreaded application, a thread in the `TPINVALIDCONTEXT` state is not allowed to issue a call to `tpresume()`.

## Return Value

`tpresume()` returns -1 on error and sets `tperrno` to indicate the error condition.

## Errors

Under the following conditions, `tpresume()` fails and sets `tperrno` to:

**[**`TPEINVAL`**]**
: Either *`tranid`* is a NULL pointer, it points to a non-existent transaction identifier (including previously completed or timed-out transactions), or it points to a transaction identifier that the caller is not allowed to resume. The caller's state with respect to the transaction is not changed.

**[**`TPEMATCH`**]**
: *`tranid`* points to a transaction identifier that another process has already resumed. The caller's state with respect to the transaction is not changed.

**[**`TPETRAN`**]**
: The BEA Tuxedo system is unable to resume the global transaction because the caller is currently participating in work outside any global transaction with one or more resource managers. All such work must be completed before a global transaction can be resumed. The caller's state with respect to the local transaction is unchanged.

**[**TPEPROTO**]**
: tpresume() was called in an improper context (for example, the caller is already in transaction mode). The caller's state with respect to the transaction is not changed.

**[**TPESYSTEM**]**
: A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPEOS**]**
: An operating system error has occurred.

### Notes

XA-compliant resource managers must be successfully opened to be included in the global transaction. (See tpopen(3c) for details.)

A process resuming a suspended transaction must reside on the same logical machine (LMID) as the process that suspended the transaction. For a Workstation client, the workstation handler (WSH) to which it is connected must reside on the same logical machine as the handler for the Workstation client that suspended the transaction.

### See Also

tpabort(3c), tpcommit(3c), tpopen(3c), tpsuspend(3c)

## tpreturn(3c)

### Name

tpreturn()—Returns from a BEA Tuxedo ATMI system service routine.

### Synopsis

```
void tpreturn(int rval, long rcode, char *data, long len, long \
  flags)
```

### Description

tpreturn() indicates that a service routine has completed. tpreturn() acts like a return statement in the C language (that is, when tpreturn() is called, the service routine returns to the BEA Tuxedo ATMI system dispatcher). It is recommended that tpreturn() be called from within the service routine dispatched to ensure correct return of control to the BEA Tuxedo ATMI system dispatcher.

tpreturn() is used to send a service's reply message. If the program receiving the reply is waiting in either tpcall(), tpgetrply(), or tprecv(), then after a successful call to tpreturn(), the reply is available in the receiver's buffer.

For conversational services, tpreturn() also tears down the connection. That is, the service routine cannot call tpdiscon() directly. To ensure correct results, the program that connected to the conversational service should not call tpdiscon(); rather, it should wait for notification that the conversational service has completed (that is, it should wait for one of the events, like TPEV_SVCSUCC or TPEV_SVCFAIL, sent by tpreturn()).

If the service routine was in transaction mode, tpreturn() places the service's portion of the transaction in a state from which it may be either committed or rolled back when the transaction is completed. A service may be invoked multiple times as part of the same transaction so it is not necessarily fully committed or rolled back until either tpcommit() or tpabort() is called by the originator of the transaction.

tpreturn() should be called after receiving all replies expected from service requests initiated by the service routine. Otherwise, depending on the nature of the service, either a TPESVCERR status or a TPEV_SVCERR event will be returned to the program that initiated communication with the service routine. Any outstanding replies that are not received will automatically be dropped by the communication manager. In addition, the descriptors for those replies become invalid.

tpreturn() should be called after closing all connections initiated by the service. Otherwise, depending on the nature of the service, either a TPESVCERR or a TPEV_SVCERR event will be returned to the program that initiated communication with the service routine. Also, an immediate disconnect event (that is, TPEV_DISCONIMM) is sent over all open connections to subordinates.

Since a conversational service has only one open connection which it did not initiate, the communication manager knows over which descriptor data (and any event) should be sent. For this reason, a descriptor is not passed to tpreturn().

The following is a description of the arguments for tpreturn(). *rval* can be set to one of the following:

TPSUCCESS
: The service has terminated successfully. If data is present, then it will be sent (barring any failures processing the return). If the caller is in transaction mode, then tpreturn() places the caller's portion of the transaction in a state such that it can be committed when the transaction ultimately commits. Note that a call to tpreturn() does not necessarily finalize an entire transaction. Also, even though the caller indicates success, if there are any outstanding replies or open connections, if any work done within the service caused its transaction to be marked rollback-only, then a failed message is sent (that is, the recipient of the reply receives a TPESVCERR indication or a TPEV_SVCERR event). Note

that if a transaction becomes rollback-only while in the service routine for any reason, then *rval* should be set to TPFAIL. If TPSUCCESS is specified for a conversational service, a TPEV_SVCSUCC event is generated.

TPFAIL
: The service has terminated unsuccessfully from an application standpoint. An error will be reported to the program receiving the reply. That is, the call to get the reply will fail and the recipient receives a TPSVCFAIL indication or a TPEV_SVCFAIL event. If the caller is in transaction mode, then tpreturn() marks the transaction as rollback-only (note that the transaction may already be marked rollback-only). Barring any failures in processing the return, the caller's data is sent, if present. One reason for not sending the caller's data is that a transaction timeout has occurred. In this case, the program waiting for the reply will receive an error of TPETIME. If TPFAIL is specified for a conversational service, a TPEV_SVCFAIL event is generated.

TPEXIT
: This value behaves the same as TPFAIL with respect to completing the service, but when TPEXIT is returned, the server exits after the transaction is rolled back and the reply is sent back to the requester.
When specified for a multithreaded process, TPEXIT indicates that an entire process (not only a single thread within that process) will be killed.
If the server is restartable, then the server is restarted automatically.

If *rval* is not set to one of these three values, then it defaults to TPFAIL.

An application defined return code, *rcode*, may be sent to the program receiving the service reply. This code is sent regardless of the setting of *rval* as long as a reply can be successfully sent (that is, as long as the receiving call returns success or TPESVCFAIL). In addition, for conversational services, this code can be sent only if the service routine has control of the connection when it issues tpreturn(). The value of *rcode* is available in the receiver in the variable, tpurcode().

*data* points to the data portion of a reply to be sent. If *data* is non-NULL, it must point to a buffer previously obtained by a call to tpalloc(). If this is the same buffer passed to the service routine upon its invocation, then its disposition is up to the BEA Tuxedo ATMI system dispatcher; the service routine writer does not have to worry about whether it is freed or not. In fact, any attempt by the user to free this buffer will fail. However, if the buffer passed to tpreturn() is not the same one with which the service is invoked, then tpreturn() frees that buffer. Although the main buffer is freed, any buffers referenced by embedded fields within that buffer are not freed. *len* specifies the amount of the data buffer to be sent. If *data* points to a buffer which does not require a length to be specified, (for example, an FML fielded buffer), then *len* is ignored (and can be 0).

If *data* is NULL, then *len* is ignored. In this case, if a reply is expected by the program that invoked the service, then a reply is sent with no data. If no reply is expected, then tpreturn() frees *data* as necessary and returns sending no reply.

Currently, *flags* is reserved for future use and must be set to 0 (if set to a non-zero value, the recipient of the reply receives a TPESVCERR indication or a TPEV_SVCERR event).

If the service is conversational, there are two cases where the caller's return code and the data portion are not transmitted:

- If the connection has already been torn down when the call is made (that is, the caller has received TPEV_DISCONIMM on the connection), then this call simply ends the service routine and rolls back the current transaction, if one exists.
- If the caller does not have control of the connection, either TPEV_SVCFAIL or TPEV_SVCERR is sent to the originator of the connection as described above. Regardless of which event the originator receives, no data is transmitted; however, if the originator receives the TPEV_SVCFAIL event, the return code is available in the originator's tpurcode() variable.

## Return Values

A service routine does not return any value to its caller, the BEA Tuxedo ATMI system dispatcher; thus, it is declared as a void. Service routines, however, are expected to terminate using either tpreturn() or tpforward(). A conversational service routine must use tpreturn(), and cannot use tpforward(). If a service routine returns without using either tpreturn() or tpforward() (that is, it uses the C language return statement or just simply "falls out of the function") or tpforward() is called from a conversational server, the server will print a warning message in the log and return a service error to the service requester. In addition, all open connections to subordinates will be disconnected immediately, and any outstanding asynchronous replies will be dropped. If the server was in transaction mode at the time of failure, the transaction is marked rollback-only. Note also that if either tpreturn() or tpforward() are used outside of a service routine (for example, in clients, or in tpsvrinit() or tpsvrdone()), then these routines simply return having no effect.

## Errors

Since tpreturn() ends the service routine, any errors encountered either in handling arguments or in processing cannot be indicated to the function's caller. Such errors cause tperrno to be set to TPESVCERR for a program receiving the service's outcome via either tpcall() or tpgetrply(), and cause the event, TPEV_SVCERR, to be sent over the conversation to a program using tpsend() or tprecv().

If either SVCTIMEOUT in the UBBCONFIG file or TA_SVCTIMEOUT in the TM_MIB is non-zero, the event TPEV_SVCERR is returned when a service timeout occurs.

tperrordetail() and tpstrerrordetail() can be used to get additional information about an error produced by the last BEA Tuxedo ATMI system routine called in the current thread. If an error occurred, tperrordetail() returns a numeric value that can be used as an argument to trstrerrordetail() to retrieve the text of the error detail.

### See Also

tpalloc(3c), tpcall(3c), tpconnect(3c), tpforward(3c), tprecv(3c), tpsend(3c), tpservice(3c)

## tpsblktime(3c)

### Name

tpsblktime() —Routine for setting blocktime in seconds for the next service call or for all service calls

### Synopsis

```
#include <atmi.h>
int tpsblktime(int blktime,long flags)
```

### Description

tpsblktime() is used to set the blocktime value, in seconds, of a potential blocking API. A *potential blocking* API is defined as: any system API that can use the flag TBNOBLOCK as a value. It does not have any effect on transaction timeout values.

The blktime range is 0 to 32767. Effective blocktime values are rounded up to the nearest multiple of the SCANUNIT value as depicted in the following example:

| User Set Blocktime Value | Scanunit Value | Effective Blocktime Value |
|---|---|---|
| 13 | 5 | 15 |
| 18 | 5 | 20 |

A 0 value indicates that any *previously set* blocking time flag value is cancelled, and the blocking time set with a different blocktime flag value prevails. If tpsblktime() is not called, the

BLOCKTIME value in the *SERVICES section or the default *RESOURCES section of the UBBCONFIG file is used.

**Note:** Blocking timeouts set with tpsblktime() take precedence over the BLOCKTIME parameter set in the SERVICES and RESOURCES section of the UBBCONFIG file. The precedence for blocktime checking is as follows:
tpsblktime(TPBLK_NEXT), tpsblktime(TPBLK_ALL), *SERVICES, *RESOURCES

The following is a list of valid flags:

TPBLK_NEXT

This flag sets the blocktime value, in seconds, for the *next* potential blocking API. Any API that is called containing the TPNOBLOCK flag is not effected by tpsblktime (TPBLK_NEXT) and continues to be non-blocking.

A TPBLK_NEXT flag value overrides a TPBLK_ALL flag value for those API calls that immediately follow it. For example:

```
tpsblktime(50,TPBLK_ALL)
tpcall(one)
tpsblktime(30,TPBLK_NEXT)
tpcall(two)
tpcall(three)
```

tpcall(two) will have a 30 second blocking timeout based on tpsblktime(30,TPBLK_NEXT). tpcall(one) and tpcall(three) will have a 50 second blocking timeout based on tpsblktime(50,TPBLK_ALL).

tpsblktime(TPBLK_NEXT) operates on a *per-thread* basis. Therefore, it is not necessary for applications to use any mutex around the tpsblktime(TPBLK_NEXT) call and the subsequent API call which it affects.

TPBLK_ALL

This flag sets the blocktime value, in seconds, for the *all* subsequent potential blocking APIs until the next tpsblktime() is called within that context. Any API that is called containing the TPNOBLOCK flag is not effected by tpsblktime(TPBLK_ALL) and continues to be non-blocking.

tpsblktime(TPBLK_ALL) operates on a *per-context* basis. Therefore, it is necessary to call tpsblktime(TPBLK_ALL) in only one thread of context that is used in multiple threads.

tpsblktime(TPBLK_ALL) will not affect any context that follows after tpterm(3c) is called.

**Note:** In order to perform blocking time values that are not affected by thread timing dependencies, it is best that `tpsblktime(TPBLK_ALL)` is called in a multi-threaded context immediately after `tpinit(3c)` using the `TPMULTICONTEXTS` flag and before the return value of `tpgetctxt(3c)` is made available to other threads.

When `tpsblktime(TPBLK_ALL)` is called in a service on a multi-threaded server, it will affect the *currently* executed thread only. To set the blocktime for all services, it is best to use `tpsblktime(TPBLK_ALL)` with `tpsvrinit(3c)` or `tpsvrthrinit(3c)`.

### Return Values

`tpsblktime()` returns -1 on error and sets `tperrno` to indicate the error condition.

### Error

Upon failure, `tpsblktime()` sets `tperrno` to one of the following values:

**[**`TPEINVAL`**]**
Invalid arguments were given. For example, the value of *`blktime`* is negative or more than one `TPBLK_NEXT` and `TPBLK_ALL` blocktime flag value is specified.

**[**`TPERELEASE`**]**
`tpsblktime()` was called in a client attached to a workstation handler running an earlier Tuxedo release.

**[**`TPESYSTEM`**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

### See Also

tpcall(3c), tpcommit(3c), tprecv(3c), tpgblktime(3c), `UBBCONFIG(5)`

## tpscmt(3c)

### Name

`tpscmt()`—Routine for setting when `tpcommit()` should return.

### Synopsis

```
#include <atmi.h>
int tpscmt(long flags)
```

## Description

tpscmt() sets the TP_COMMIT_CONTROL characteristic to the value specified in *flags*. The TP_COMMIT_CONTROL characteristic affects the way tpcommit() behaves with respect to returning control to its caller. A program can call tpscmt() regardless of whether it is in transaction mode or not. Note that if the caller is participating in a transaction that another program must commit, then its call to tpscmt() does not affect that transaction. Rather, it affects subsequent transactions that the caller will commit.

In most cases, a transaction is committed only when a BEA Tuxedo ATMI system thread of control calls tpcommit(). There is one exception: when a service is dispatched in transaction mode because the AUTOTRAN variable in the *SERVICES section of the UBBCONFIG file is enabled, then the transaction completes upon calling tpreturn(). If tpforward() is called, then the transaction will be completed by the server ultimately calling tpreturn(). Thus, the setting of the TP_COMMIT_CONTROL characteristic in the service that calls tpreturn() determines when tpcommit() returns control within a server. If tpcommit() returns a heuristic error code, the server will write a message to a log file.

When a client joins a BEA Tuxedo ATMI system application, the initial setting for this characteristic comes from a configuration file. (See the CMTRET variable in the RESOURCES section of UBBCONFIG(5))

The following are the valid settings for *flags*:

TP_CMT_LOGGED
: This flag indicates that tpcommit() should return after the commit decision has been logged by the first phase of the two-phase commit protocol but before the second phase has completed. This setting allows for faster response to the caller of tpcommit() although there is a risk that a transaction participant might decide to heuristically complete (that is, abort) its work due to timing delays waiting for the second phase to complete. If this occurs, there is no way to indicate this situation to the caller since tpcommit() has already returned (although the BEA Tuxedo ATMI system writes a message to a log file when a resource manager takes a heuristic decision). Under normal conditions, participants that promise to commit during the first phase will do so during the second phase. Typically, problems caused by network or site failures are the sources for heuristic decisions being made during the second phase.

TP_CMT_COMPLETE
: This flag indicates that tpcommit(3c) should return after the two-phase commit protocol has finished completely. This setting allows for tpcommit() to return an indication that a heuristic decision occurred during the second phase of commit.

In a multi-threaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tpscmt().

### Return Values

Upon success, tpscmt() returns the previous value of the TP_COMMIT_CONTROL characteristic.

Upon failure, tpscmt() returns -1 and sets tperrno to indicate the error condition.

### Errors

Upon failure, tpscmt() sets tperrno to one of the following values:

**[**TPEINVAL**]**
*flags* is not one of TP_CMT_LOGGED or TP_CMT_COMPLETE.

**[**TPEPROTO**]**
tpscmt() was called improperly.

**[**TPESYSTEM**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPEOS**]**
An operating system error has occurred.

### Notices

When using tpbegin(), tpcommit() and tpabort() to delineate a BEA Tuxedo ATMI system transaction, it is important to remember that only the work done by a resource manager that meets the XA interface (and is linked to the caller appropriately) has transactional properties. All other operations performed in a transaction are not affected by either tpcommit() or tpabort(). See buildserver(1) for details on linking resource managers that meet the XA interface into a server such that operations performed by that resource manager are part of a BEA Tuxedo ATMI system transaction.

### See Also

tpabort(3c), tpbegin(3c), tpcommit(3c), tpgetlev(3c)

## tpseal(3c)

### Name

tpseal()—Marks a typed message buffer for encryption.

### Synopsis

```
#include <atmi.h>
int tpseal(char *data, TPKEY hKey, long flags)
```

### Description

`tpseal()` marks, or registers, a message buffer for encryption. The principal who owns *`hKey`* can decrypt this buffer and access its content. A buffer may be sealed for more than one recipient principal by making several calls to `tpseal()`.

*`data`* must point to a valid typed message buffer either (1) previously allocated by a process calling `tpalloc()` or (2) delivered by the system to a receiving process. The content of the buffer may be modified after `tpseal()` is invoked.

When the message buffer pointed to by *`data`* is transmitted from a process, the public key software encrypts the message content and attaches an encryption envelope to the message buffer for each encryption registration request. An encryption envelope enables a receiving process to decrypt the message.

The *`flags`* argument is reserved for future use and must be set to 0.

### Return Values

On failure, this function returns -1 and sets `tperrno` to indicate the error condition.

### Errors

**[**`TPEINVAL`**]**
: Invalid arguments were given. For example, *`hKey`* is not a valid key for encrypting or *`data`* is NULL.

**[**`TPESYSTEM`**]**
: An error has occurred. Consult the system error log file for details.

### See Also

`tpkey_close(3c)`, `tpkey_open(3c)`

## tpsend(3c)

### Name

`tpsend()`—Routine for sending a message in a conversational connection.

## Synopsis

```
#include <atmi.h>
int tpsend(int cd, char *data, long len, long flags, long *revent)
```

## Description

`tpsend()` is used to send data across an open connection to another program. The caller must have control of the connection. `tpsend()`'s first argument, *cd*, specifies the open connection over which data is sent. *cd* is a descriptor returned from either `tpconnect()` or the `TPSVCINFO` parameter passed to a conversational service.

The second argument, *data*, must point to a buffer previously allocated by `tpalloc()`. *len* specifies how much of the buffer to send. Note that if *data* points to a buffer of a type that does not require a length to be specified (for example, an `FML` fielded buffer), then *len* is ignored (and may be 0). Also, *data* can be NULL in which case *len* is ignored (no application data is sent—this might be done, for instance, to grant control of the connection without transmitting any data). The type and subtype of *data* must match one of the types and subtypes recognized by the other end of the connection.

The following is a list of valid *flags*:

`TPRECVONLY`
: This flag signifies that, after the caller's data is sent, the caller gives up control of the connection (that is, the caller can not issue any more `tpsend`() calls). When the receiver on the other end of the connection receives the data sent by `tpsend()`, it will also receive an event (`TPEV_SENDONLY`) indicating that it has control of the connection (and can not issue more any `tprecv()` calls).

`TPNOBLOCK`
: The data and any events are not sent if a blocking condition exists (for example, the internal buffers into which the message is transferred are full). When `TPNOBLOCK` is not specified and a blocking condition exists, the caller blocks until the condition subsides or a timeout occurs (either transaction or blocking timeout).

`TPNOTIME`
: This flag signifies that the caller is willing to block indefinitely and wants to be immune to blocking timeouts. Transaction timeouts may still occur.

`TPSIGRSTRT`
: If a signal interrupts any underlying system calls, then the interrupted system call is reissued.

If an event exists for the descriptor, *cd*, then `tpsend()` will fail without sending the caller's data. The event type is returned in *revent*. Valid events for `tpsend()` are as follows:

TPEV_DISCONIMM
: Received by the subordinate of a conversation, this event indicates that the originator of the conversation has issued an immediate disconnect on the connection via tpdiscon(), or it issued tpreturn(), tpcommit() or tpabort() with the connection still open. This event is also returned to the originator or subordinate when a connection is broken due to a communications error (for example, a server, machine, or network failure).

TPEV_SVCERR
: Received by the originator of a conversation, this event indicates that the subordinate of the conversation has issued tpreturn() without having control of the conversation. In addition, tpreturn() has been issued in a manner different from that described for TPEV_SVCFAIL below. This event can be caused by an ACL permissions violation; that is, the originator does not have permission to connect to the receiving process. This event is not returned at the time the tpconnect() is issued, but is returned with the first tpsend() (following a tpconnect() with flag TPSENDONLY) or tprecv() (following a tpconnect() with flag TPRECVONLY). A system event and a log message are also generated.

TPEV_SVCFAIL
: Received by the originator of a conversation, this event indicates that the subordinate of the conversation has issued tpreturn() without having control of the conversation. In addition, tpreturn() was issued with the *rval* set to TPFAIL or TPEXIT and *data* to NULL.

Because each of these events indicates an immediate disconnection notification (that is, abortive rather than orderly), data in transit may be lost. The descriptor used for the connection is no longer valid. If the two programs were participating in the same transaction, then the transaction has been marked abort-only.

If the value of either SVCTIMEOUT in the UBBCONFIG file or TA_SVCTIMEOUT in the TM_MIB is non-zero, TPESVCERR is returned when a service timeout occurs.

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tpsend().

## Return Values

Upon return from tpsend() where *revent* is set to either TPEV_SVCSUCC or TPEV_SVCFAIL, the tpurcode() global contains an application-defined value that was sent as part of tpreturn(). The function tpsend() returns -1 on error and sets tperrno to indicate the error condition. Also, if an event exists and no errors were encountered, tpsend() returns -1 and tperrno is set to [TPEEVENT].

## Errors

Upon failure, `tpsend()` sets `tperrno` to one of the following values:

**[**TPEINVAL**]**
Invalid arguments were given (for example, *data* does not point to a buffer allocated by `tpalloc()` or *flags* are invalid).

**[**TPEBADDESC**]**
*cd* is invalid.

**[**TPETIME**]**
This error code indicates that either a timeout has occurred or `tpsend()` has been attempted, in spite of the fact that the current transaction is already marked rollback only.

If the caller is in transaction mode, then either the transaction is already rollback only or a transaction timeout has occurred. The transaction is marked abort-only. If the caller is not in transaction mode, a blocking timeout has occurred. (A blocking timeout cannot occur if `TPNOBLOCK` and/or `TPNOTIME` is specified.)

If a transaction timeout has occurred, then, with one exception, any attempts to send new requests or receive outstanding replies will fail with `TPETIME` until the transaction has been aborted. The exception is a request that does not block, expects no reply, and is not sent on behalf of the caller's transaction (that is, `tpacall()` with `TPNOTRAN`, `TPNOBLOCK`, and `TPNOREPLY` set).

When a transactional ATMI call fails inside a transaction, the transaction is put into the `TX_ROLLBACK_ONLY` state. This state is treated, for most purposes, as though it were equivalent to a timeout. All further ATMI calls for this transaction (with the exception of those issued in the circumstances described in the previous paragraph) will fail with `TPETIME`.

**[**TPEEVENT**]**
An event occurred. *data* is not sent when this error occurs. The event type is returned in *revent*.

**[**TPEBLOCK**]**
A blocking condition exists and `TPNOBLOCK` was specified.

**[**TPGOTSIG**]**
A signal was received and `TPSIGRSTRT` was not specified.

**[**TPEPROTO**]**
`tpsend()` was called in an improper context (for example, the connection was established such that the calling program can only receive data).

**[**TPESYSTEM**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPEOS**]**
An operating system error has occurred.

### See Also

tpalloc(3c), tpconnect(3c), tpdiscon(3c), tprecv(3c), tpservice(3c)

## tpservice(3c)

### Name

tpservice()—Template for service routines.

### Synopsis

```
#include <atmi.h>                    /* C interface */
void tpservice(TPSVCINFO *svcinfo)   /* C++ interface - must have
                                      * C linkage */
extern "C" void tpservice(TPSVCINFO *svcinfo)
```

### Description

tpservice() is the template for writing service routines. This template is used for services that receive requests via tpcall(), tpacall() or tpforward() routines as well as by services that communicate via tpconnect(), tpsend() and tprecv() routines.

Service routines processing requests made via either tpcall() or tpacall() receive at most one incoming message (in the *data* element of *svcinfo*) and send at most one reply (upon exiting the service routine with tpreturn()).

Conversational services, on the other hand, are invoked by connection requests with at most one incoming message along with a means of referring to the open connection. When a conversational service routine is invoked, either the connecting program or the conversational service may send and receive data as defined by the application. The connection is half-duplex in nature meaning that one side controls the conversation (that is, it sends data) until it explicitly gives up control to the other side of the connection.

Concerning transactions, service routines can participate in at most one transaction if invoked in transaction mode. As far as the service routine writer is concerned, the transaction ends upon returning from the service routine. If the service routine is not invoked in transaction mode, then

the service routine may originate as many transactions as it wants using `tpbegin()`, `tpcommit()`, and `tpabort()`. Note that `tpreturn()` is not used to complete a transaction. Thus, it is an error to call `tpreturn()` with an outstanding transaction that originated within the service routine.

Service routines are invoked with one argument: *svcinfo*, a pointer to a service information structure. This structure includes the following members:

```
char        name[32];
char        *data;
long        len;
long        flags;
int         cd;
long        appkey;
CLIENTID     cltid;
```

`name` is populated with the service name that the requester used to invoke the service.

The setting of *flags* upon entrance to a service routine indicates attributes which the service routine may want to note. The following are the possible values for *flags*:

`TPCONV`
: A connection request for a conversation has been accepted and the descriptor for the conversation is available in *cd*. If not set, then this is a request/response service and *cd* is not valid.

`TPTRAN`
: The service routine is in transaction mode.

`TPNOREPLY`
: The caller is not expecting a reply. This option will not be set if `TPCONV` is set.

`TPSENDONLY`
: The service is invoked such that it can only send data across the connection and the program on the other end of the connection can only receive data. This flag is mutually exclusive with `TPRECVONLY` and may be set only when `TPCONV` is also set.

`TPRECVONLY`
: The service is invoked such that it can only receive data from the connection and the program on the other end of the connection can only send data. This flag is mutually exclusive with `TPSENDONLY` and may be set only when `TPCONV` is also set.

*data* points to the data portion of a request message and *len* is the length of the data. The buffer pointed to by *data* was allocated by `tpalloc()` in the communication manager. This buffer may

be grown by the user with tprealloc(); however, it cannot be freed by the user. It is recommended that this buffer be the one passed to either tpreturn() or tpforward() when the service ends. If a different buffer is passed to those routines, then that buffer is freed by them. Note that the buffer pointed to by *data* will be overwritten by the next service request even if this buffer is not passed to tpreturn() or tpforward(). *data* may be NULL if no data accompanied the request. In this case, *len* will be 0.

When TPCONV is set in *flags*, *cd* is the connection descriptor that can be used with tpsend() and tprecv() to communicate with the program that initiated the conversation.

*appkey* is set to the application key assigned to the requesting client by the application defined authentication service. This key value is passed along with any and all service requests made while within this invocation of the service routine. *appkey* will have a value of -1 for originating clients that do not pass through the application authentication service.

*cltid* is the unique client identifier for the originating client associated with this service request. The definition of this structure is made available to the application in atmi.h solely so that client identifiers may be passed between application servers if necessary. Therefore, the semantics of the fields defined below are not documented and applications should not manipulate the contents of CLIENTID structures. Doing so will invalidate the structures. The CLIENTID structure includes the following member:

```
long        clientdata[4];
```

Note that for C++, the service function must have C linkage. This is done by declaring the function as ‘extern “C.”’

## Return Values

A service routine does not return any value to its caller, the communication manager dispatcher; thus, it is declared as a void. Service routines, however, are expected to terminate using either tpreturn() or tpforward(). A conversational service routine must use tpreturn(), and cannot use tpforward(). If a service routine returns without using either tpreturn() or tpforward() (that is, it uses the C language return statement or just simply “falls out of the function”) or tpforward() is called from a conversational server, the server will print a warning message in a log file and return a service error to the originator or requester. All open connections to subordinates will be disconnected immediately, and any outstanding asynchronous replies will be marked stale. If the server was in transaction mode at the time of failure, the transaction is marked abort-only. Note also that if either tpreturn() or tpforward() are used outside of a service routine (for example, in clients, or in tpsvrinit() or tpsvrdone()), then these routines simply return having no effect.

### Errors

Since tpreturn() ends the service routine, any errors encountered either in handling arguments or in processing cannot be indicated to the function's caller. Such errors cause tperrno to be set to TPESVCERR for a program receiving the service's outcome via either tpcall() or tpgetrply(), and cause the event, TPEV_SVCERR, to be sent over the conversation to a program using tpsend() or tprecv().

### See Also

tpalloc(3c), tpbegin(3c), tpcall(3c), tpconnect(3c), tpforward(3c), tpreturn(3c), servopts(5)

## tpsetctxt(3c)

### Name

tpsetctxt()—Sets a context identifier for the current application association.

### Synopsis

```
#include <atmi.h>
int tpsetctxt(TPCONTEXT_T context, long flags)
```

### Description

tpsetctxt() defines the context in which the current thread operates. This function operates on a per-thread basis in a multithreaded environment, and on a per-process basis in a non-threaded environment.

Subsequent BEA Tuxedo ATMI calls made in this thread reference the application indicated by context. The context should have been provided by a previous call to tpgetctxt() in one of the threads of the same process. If the value of *context* is TPNULLCONTEXT, then the current thread is disassociated from any BEA Tuxedo ATMI context.

You can put an individual thread in a process operating in multicontext mode into the TPNULLCONTEXT state by issuing the following call:

```
tpsetctxt(TPNULLCONTEXT, 0)
```

TPINVALIDCONTEXT is not a valid input value for *context*.

A thread in the TPINVALIDCONTEXT state is prohibited from issuing calls to most ATMI functions. (For a complete list of the functions that may and may not be called, see "Introduction to the C Language Application-to-Transaction Monitor Interface.".) Therefore, you may

sometimes need to move a thread out of the TPINVALIDCONTEXT state. To do so, call tpsetctxt() with context set to TPNULLCONTEXT or another valid context. (It is also allowable to call the tpterm() function to exit from the TPINVALIDCONTEXT state.)

The second argument, *flags*, is not currently used and must be set to 0.

A thread in a multithreaded application may issue a call to tpsetctxt() while running in any context state, including TPINVALIDCONTEXT.

## Return Values

Upon successful completion, tpsetctxt() returns a non-negative value.

Upon failure, it leaves the calling process in its original context, returns a value of -1, and sets tperrno to indicate the error condition.

## Errors

Upon failure, tpsetctxt() sets tperrno to one of the following values:

**[**TPEINVAL**]**
: Invalid arguments have been given. For example, *flags* has been set to a value other than 0 or the context is TPINVALIDCONTEXT.

**[**TPENOENT**]**
: The value of *context* is not a valid context.

**[**TPEPROTO**]**
: tpsetctxt() has been called in an improper context. For example: (a) it has been called in a server-dispatched thread; (b) it has been called in a process that has not called tpinit(); (c) it has been called in a process that has called tpinit() without specifying the TPMULTICONTEXTS flag; or (d) it has been called from more than one thread in a process where the TMNOTHREADS environment variable has been turned on.

**[**TPESYSTEM**]**
: A BEA Tuxedo system error has occurred. The exact nature of the error has been written to a log file.

**[**TPEOS**]**
: An operating system error has occurred.

## See Also

Introduction to the C Language Application-to-Transaction Monitor Interface, tpgetctxt(3c)

# tpsetmbenc(3c)

## Name

`tpsetmbenc()`—Sets the code-set encoding name for a typed buffer.

## Synopsis

```
#include <atmi.h>
extern int tperrno;

int
tpsetmbenc(char *bufp, char *enc_name, long flags)
```

## Description

This function is used for setting or resetting the codeset encoding name. The encoding name is sent along with the input typed buffer. A process receiving these message can use `tpgetmbenc()` to retrieve this encoding name.

`tpsetmbenc()` sets the codeset encoding name to be included with a Tuxedo system request. Once this function sets a non-NULL encoding name in the caller's buffer, all requests sent (via `tpcall()`, `tpsend()`) include this string until reset or unset. An initial codeset encoding name is applied to a MBSTRING buffer, during `tpalloc()`, using the TPMBENC environment variable. An MBSTRING buffer without an encoding name defined is invalid.

The *`bufp`* argument is a valid pointer to a typed buffer with an encoding name.

The *`enc_name`* argument is the encoding name to use to identify the codeset encoding.

The *`flags`* argument is 0 or RM_ENC. For RM_ENC the encoding name will be removed from the MBSTRING buffer and the *`enc_name`* argument will be ignored. Note that an MBSTRING buffer without an encoding name will fail the _tmconvmb() conversion.

## Return Values

Upon success, `tpsetmbenc()` returns a 0 value otherwise it returns a non-zero on error and sets `tperrno` to indicate the error condition. This function may fail for the following reasons.

**[**`TPEINVAL`**]**
*`buf`*, *`enc_name`* argument is NULL or *`enc_name`* is not a valid name to use.

**[**`TPESYSTEM`**]**
A Tuxedo system error has occurred. (e.g. bufp does not correspond to a valid Tuxedo buffer, could not add or remove the encoding name from the buffer)

### See Also

tpalloc(3c), tpconvmb(3c), tpgetmbenc(3c), tpservice(3c), tuxsetmbenc(3c)

## tpsetrepos(3c)

### Name

tpsetrepos() - adds, edits, or deletes service parameter information from a Tuxedo Service Metadata repository file

### Synopsis

```
int tpsetrepos(char *reposfile, FBFR32* idata, FBFR32** odata)
```

### Description

tpsetrepos() provides an alternative repository access interface to the .TMMETAREPOS service provided by TMMETADATA(5). It adds, edits, or deletes parameter information from a Tuxedo Service Metadata repository file. To use tpsetrepos(), the metadata repository file must reside on the native client or server that initiates the request. This allows for repository information access even when TMMETADATA(5) has not been booted.

tpsetrepos() is available in processes linked with the BEA Tuxedo native libraries, but is not available in processes linked with the BEA Tuxedo workstation libraries.

**Note:** tpsetrepos() cannot be used to add, edit, or delete service parameter information in a JOLT Repository file.

reposfile
: specifies the path name of a file accessible on the current machine where the Tuxedo Metadata Repository is located. The user must have read and write permissions for this file.

idata
: specifies what type of service information is added, edited, or deleted, and points to an FML32 buffer.

***odata**
: On output, points to an FML32 buffer containing the retrieved service information and operation status.

METAREPOS(5) describes the FML32 buffer format tpsetrepos()uses. It is similar to the format used by MIB(5).

## Return Values

`tpsetrepos()` returns 0 on success. On failure, it sets tperrno and returns -1. On most failure conditions, the `TA_ERROR` field in `*odata` is populated with information about the specific error, as is done by the Tuxedo MIB.

## Errors

Upon failure, `tpsetrepos()` sets `tperrno` to one of the following values:

**Note:** Except for `TPEINVAL`, `odata` is modified to include `TA_ERROR`, `TA_STATUS` for each service entry to further qualify the error condition.

`[TPEINVAL]`
Invalid arguments were specified. The reposfile value is invalid or `idata` or `odata` are not pointers to `FML32` typed buffers.

`[TPEMIB]`
The `MIB`-like request failed. odata is updated and returned to the caller with `FML32` fields indicating the cause of the error as discussed in `MIB(5)`.

`[TPEPROTO]`
`tpsetrepos()` was improperly called. The reposfile file argument given is not a valid repository file.

`[TPEPERM]`
A Jolt repository file is specified. `tpsetrepos()` cannot be applied to a Jolt repository file.

`[TPEOS]`
An operating system error has occurred. A numeric value representing the system call that failed is available in `Uunixerr`.

`[TPESYSTEM]`
A BEA Tuxedo system error has occurred. The exact nature of the error is reported in `userlog()`.

## Portability

This interface is available only on BEA Tuxedo release 9.0 or later.

## Files

The following library files are required:

```
${TUXDIR}/lib/libtrep.a
${TUXDIR}/lib/libtrep.so.<rel>
${TUXDIR}/lib/libtrep.lib
```

The libraries must be linked manually when using buildclient. The user must use:
`-L${TUXDIR}/lib -ltrep`

### See Also

`tpgetrepos(3c)`, `tmloadrepos(1)`, `tmunloadrepos(1)`, `TMMETADATA(5)`, Managing The Tuxedo Service Metadata Repository

## tpsetunsol(3c)

### Name

`tpsetunsol()`—Sets the method for handling unsolicited messages.

### Synopsis

```
#include <atmi.h>
void (*tpsetunsol (void (_TMDLLENTRY *)(*disp) (char *data, long len, long
flags))) (char *data, long len, long flags)
```

### Description

`tpsetunsol()` allows a client to identify the routine that should be invoked when an unsolicited message is received by the BEA Tuxedo ATMI system libraries. Before the first call to `tpsetunsol()`, any unsolicited messages received by the BEA Tuxedo ATMI system libraries on behalf of the client are logged and ignored. A call to `tpsetunsol()` with a NULL function pointer has the same effect. The method used by the system for notification and detection is determined by the application default, which can be overridden on a per-client basis (see `tpinit(3c)`).

The function pointer passed on the call to `tpsetunsol()` must conform to the parameter definition given. The `_TMDLLENTRY` macro is required for Windows-based operating systems to obtain the proper calling conventions between the Tuxedo libaries and your code. On Unix systems, the `_TMDLLENTRY` macro is not required because it expands to the null string.

*`data`* points to the typed buffer received and *`len`* is the length of the data. *`flags`* are currently unused. *`data`* can be NULL if no data accompanied the notification. *`data`* may be of a buffer type/subtype that is not known by the client, in which case the message data is unintelligible.

*`data`* cannot be freed by application code. However, the system frees it and invalidates the data area following return.

Processing within the application's unsolicited message handling routine is restricted to the following BEA Tuxedo ATMI functions: `tpalloc()`, `tpfree()`, `tpgetctxt()`, `tpgetlev()`, `tprealloc()`, and `tptypes()`.

Note that in a multithreaded programming environment, it is possible for an unsolicited message handling routine to call `tpgetctxt()`, create another thread, have that thread call `tpsetctxt()` to the appropriate context, and have the new thread use the full set of ATMI functions that are available to clients.

If `tpsetunsol()` is called from a thread that is not currently associated with a context, this establishes a per-process default unsolicited message handler for all new `tpinit()` contexts created. It has no effect on contexts already associated with the system. A specific context may change this default unsolicited message handler by calling `tpsetunsol()` again when the context is active. The per-process default unsolicited message handler may be changed by again calling `tpsetunsol()` in a thread not currently associated with a context.

In a multithreaded application, a thread in the `TPINVALIDCONTEXT` state is not allowed to issue a call to `tpsetunsol()`.

## Return Values

Upon success, `tpsetunsol()` returns the previous setting for the unsolicited message handling routine. (NULL is a successful return indicating that no message handling function had been set previously.)

Upon failure, it returns `TPUNSOLERR` and sets `tperrno` to indicate the error condition.

## Errors

Upon failure, `tpsetunsol()` sets `tperrno` to one of the following values:

**[**`TPEPROTO`**]**
: `tpsetunsol()` has been called in an improper context. For example, it has been called from within a server.

**[**`TPESYSTEM`**]**
: A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**`TPEOS`**]**
: An operating system error has occurred.

### Portability

The interfaces described in `tpnotify(3c)` are supported on native site UNIX-based and Windows processors. In addition, the routines `tpbroadcast()` and `tpchkunsol()`, as well as the function `tpsetunsol()`, are supported on UNIX and MS-DOS workstation processors.

### See Also

`tpinit(3c)`, `tpterm(3c)`

## tpsign(3c)

### Name

`tpsign()`—Marks a typed message buffer for digital signature.

### Synopsis

```
#include <atmi.h>
int tpsign(char *data, TPKEY hKey, long flags)
```

### Description

`tpsign()` marks, or registers, a message buffer for digital signature on behalf of the principal associated with *`hKey`*.

*`data`* must point to a valid typed message buffer either (1) previously allocated by a process calling `tpalloc()` or (2) delivered by the system to a receiving process. The content of the buffer may be modified after `tpsign()` is invoked.

When the buffer pointed to by *`data`* is transmitted from a process, the public key software generates and attaches a digital signature to the message buffer for each digital-signature registration request. A digital signature enables a receiving process to verify the signer (originator) of the message.

The *`flags`* argument is reserved for future use and must be set to 0.

### Return Values

On failure, this function returns -1 and sets `tperrno` to indicate the error condition.

### Errors

**[**TPEINVAL**]**
Invalid arguments were given. For example, *hKey* is not a valid key for signing or the value of *data* is NULL.

**[**TPESYSTEM**]**
An error occurred. Consult the system error log file for details.

### See Also

tpkey_close(3c), tpkey_open(3c)

## tpsprio(3c)

### Name

tpsprio()—Sets the service request priority.

### Synopsis

```
#include <atmi.h>
int tpsprio(prio, flags)
```

### Description

tpsprio() sets the priority for the next request sent or forwarded by the current thread in the current context. The priority set affects only the next request sent. Priority can also be set for messages enqueued or dequeued by tpenqueue() or tpdequeue(), if the queued message facility is installed. By default, the setting of *prio* increments or decrements a service's default priority up to a maximum of 100 or down to a minimum of 1, depending on its sign, where 100 is the highest priority. The default priority for a request is determined by the service to which the request is being sent. This default may be specified administratively (see UBBCONFIG(5)), or take the system default of 50. tpsprio() has no effect on messages sent via tpconnect() or tpsend().

A lower priority message does not remain enqueued forever because every tenth message is retrieved on a "first in, first out" (FIFO) basis. Response time should not be a concern of the lower priority interface or service.

In a multithreaded application tpsprio() operates on a per-thread basis.

The following is a list of valid flags:

TPABSOLUTE
: The priority of the next request should be sent out at the absolute value of *prio*. The absolute value of *prio* must be within the range 1 and 100, inclusive, with 100 being the highest priority. Any value outside of this range causes a default value to be used.

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tpsprio().

### Return Values

Upon failure, tpsprio() returns -1 and sets tperrno to indicate the error condition.

### Errors

Upon failure, tpsprio() sets tperrno to one of the following values:

**[**TPEINVAL**]**
: *flags* are invalid.

**[**TPEPROTO**]**
: tpsprio() was called improperly.

**[**TPESYSTEM**]**
: A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPEOS**]**
: An operating system error has occurred.

### See Also

tpacall(3c), tpcall(3c), tpdequeue(3c), tpenqueue(3c), tpgprio(3c)

## tpstrerror(3c)

### Name

tpstrerror()—Gets error message string for a BEA Tuxedo ATMI system error.

### Synopsis

```
#include <atmi.h>
char *
tpstrerror(int err)
```

## Description

`tpstrerror()` is used to retrieve the text of an error message from `LIBTUX_CAT`. *`err`* is the error code set in `tperrno` when a BEA Tuxedo ATMI system function call returns a -1 or other failure value.

You can use the pointer returned by `tpstrerror()` as an argument to `userlog()` or the UNIX function `fprintf()`.

A thread in a multithreaded application may issue a call to `tpstrerror()` while running in any context state, including `TPINVALIDCONTEXT`.

## Return Values

Upon success, `tpstrerror()` returns a pointer to a string that contains the error message text.

If *`err`* is an invalid error code, `tpstrerror()` returns a NULL.

## Errors

Upon failure, `tpstrerror()` returns a NULL but does not set `tperrno`.

## Example

```
#include <atmi.h>
.
.
.
  char *p;
  if (tpbegin(10,0) == -1) {
    p = tpstrerror(tperrno);
    userlog("%s", p);
    (void)tpabort(0);
    (void)tpterm();
    exit(1);
  }
```

## See Also

`userlog(3c),Fstrerror, Fstrerror32(3fml)`

## tpstrerrordetail(3c)

### Name

tpstrerrordetail()—Gets error detail message string for a BEA Tuxedo ATMI system error.

### Synopsis

```
#include <atmi.h>
char * tpstrerrordetail(int err, long flags)
```

### Description

tpstrerrordetail() is used to retrieve the text of an error detail of a BEA Tuxedo ATMI system error. *err* is the value returned by tperrordetail().

The user can use the pointer returned by tpstrerrordetail() as an argument to userlog() or the UNIX function fprintf().

Currently *flags* is reserved for future use and must be set to 0.

A thread in a multithreaded application may issue a call to tpstrerrordetail() while running in any context state, including TPINVALIDCONTEXT.

### Return Values

Upon success, the function returns a pointer to a string that contains the error detail message text.

Upon failure (that is, if *err* is an invalid error code), tpstrerrordetail() returns a NULL.

### Errors

Upon failure, tpstrerrordetail() returns a NULL but does not set tperrno.

### Example

```
#include <atmi.h> .  .  .
int ret;
char *p;

if (tpbegin(10,0) == -1) {
       ret = tperrordetail(0);
       if (ret == -1) {
              (void) fprintf(stderr, "tperrordetail() failed!\n");
              (void) fprintf(stderr, "tperrno = %d, %s\n",
```

```
                        tperrno, tpstrerror(tperrno));
        }

        else if (ret != 0) {
                (void) fprintf(stderr, "errordetail:%s\n",
                                tpstrerrordetail(ret, 0));
        }
        .
        .
        .
}
```

### See Also

Introduction to the C Language Application-to-Transaction Monitor Interface, `tperrordetail(3c)`, `tpstrerror(3c)`, `userlog(3c)`, `tperrno(5)`

## tpsubscribe(3c)

### Name

`tpsubscribe()`—Subscribes to an event.

### Synopsis

```
#include <atmi.h>
long tpsubscribe(char *eventexpr, char *filter, TPEVCTL *ctl, long flags)
```

### Description

The caller uses `tpsubscribe()` to subscribe to an event or set of events named by *`eventexpr`*. Subscriptions are maintained by the BEA Tuxedo ATMI EventBroker, `TMUSREVT(5)`, and are used to notify subscribers when events are posted via `tppost()`. Each subscription specifies a notification method which can take one of three forms: client notification, service calls, or message enqueuing to stable-storage queues. Notification methods are determined by the subscriber's process type and the arguments passed to `tpsubscribe()`.

The event or set of events being subscribed to is named by *`eventexpr`*, a NULL-terminated string of at most 255 characters containing a regular expression. For example, if *`eventexpr`* is "`\e\e..*`", the caller is subscribing to all system-generated events; if *`eventexpr`* is "`\e\e.SysServer.*`", the caller is subscribing to all system-generated events related to servers.

If *eventexpr* is "[A-Z].*", the caller is subscribing to all user events starting with A-Z; if *eventexpr* is ".*(ERR|err).*", the caller is subscribing to all user events containing either the substring ERR or the substring err in the event name. Events called account_error and ERROR_STATE, for example, would both qualify. For more information on regular expressions, see "Regular Expressions" on page 228.

If present, *filter* is a string containing a Boolean filter rule that must be evaluated successfully before the EventBroker posts the event. Upon receiving an event to be posted, the EventBroker applies the filter rule, if one exists, to the posted event's data. If the data passes the filter rule, the EventBroker invokes the notification method; otherwise, the broker does not invoke the associated notification method. The caller can subscribe to the same event multiple times with different filter rules.

Filter rules are specific to the typed buffers to which they are applied. For FML and view buffers, the filter rule is a string that can be passed to each's Boolean expression compiler (see Fboolco(3fml) and Fvboolco(3fml), respectively) and evaluated against the posted buffer (see Fboolev(3fml) and Fvboolev(3fml), respectively). For STRING buffers, the filter rule is a regular expression. All other buffer types require customized filter evaluators (see buffer(3c) and typesw(5) for details on adding customized filter evaluators). *filter* is a NULL-terminated string of at most 255 characters.

If the subscriber is a BEA Tuxedo ATMI system client process and *ctl* is NULL, then the EventBroker sends an unsolicited message to the subscriber when the event to which it subscribed is posted. That is, when an event name is posted that evaluates successfully against *eventexpr*, the EventBroker tests the posted data against the filter rule associated with *eventexpr*. If the data passes the filter rule or if there is no filter rule for the event, then the subscriber receives an unsolicited notification along with any data posted with the event. In order to receive unsolicited notifications, the client must register (via tpsetunsol()) an unsolicited message handling routine. If a BEA Tuxedo ATMI system server process calls tpsubscribe() with a NULL *ctl* parameter, then tpsubscribe() fails setting tperrno to TPEPROTO.

Clients receiving event notification via unsolicited messages should remove their subscriptions from the EventBroker's list of active subscriptions before exiting (see tpunsubscribe(3c) for details). Using tpunsubscribe()'s wildcard handle, -1, clients can conveniently remove all of their "non-persistent" subscriptions which include those associated with the unsolicited notification method (see the description of TPEVPERSIST below for subscriptions and their associated notification methods that persist after a process exits). If a client exits without removing its non-persistent subscriptions, then the EventBroker will remove them when it detects that the client is no longer accessible.

If the subscriber (regardless of process type) wants event notifications to go to service routines or to stable-storage queues, then the *ctl* parameter must point to a valid TPEVCTL structure. This structure contains the following elements:

```
long    flags;
char    name1[32];
char    name2[32];
TPQCTL  qctl;
```

**Note:** The service name length limit is 15 bytes. If the service name length exceeds 15 bytes, TPEINVAL is returned.

The following is a list of valid bits for the *ctl*→*flags* element controlling options for event subscriptions:

TPEVSERVICE

Setting this flag indicates that the subscriber wants event notifications to be sent to the BEA Tuxedo ATMI system service routine named in *ctl*→*name1*. That is, when an event name is posted that evaluates successfully against *eventexpr*, the EventBroker tests the posted data against the filter rule associated with *eventexpr*. If the data passes the filter rule or if there is no filter rule for the event, then a service request is sent to *ctl*→*name1* along with any data posted with the event. The service name in *ctl*→*name1* can be any valid BEA Tuxedo ATMI system service name and it may or may not be active at the time the subscription is made. Service routines invoked by the EventBroker should return with no reply data. That is, they should call tpreturn() with a NULL data argument. Any data passed to tpreturn() will be dropped. TPEVSERVICE and TPEVQUEUE are mutually exclusive flags.

If TPEVTRAN is also set in *ctl*→*flags*, then if the process calling tppost() is in transaction mode, the EventBroker calls the subscribed service routine such that it will be part of the poster's transaction. Both the EventBroker, TMUSREVT(5), and the subscribed service routine must belong to server groups that support transactions (see UBBCONFIG(5) for details). If TPEVTRAN is not set in *ctl*→*flags*, then the EventBroker calls the subscribed service routine such that it will not be part of the poster's transaction.

TPEVQUEUE

Setting this flag indicates that the subscriber wants event notifications to be enqueued to the queue space named in *ctl*→*name1* and the queue named in *ctl*→*name2*. That is, when an event name is posted that evaluates successfully against *eventexpr*, the EventBroker tests the posted data against the filter rule associated with *eventexpr*. If the data passes the filter rule or if there is no filter rule for the event, then the EventBroker enqueues a message to the queue space named in *ctl*→*name1* and the queue named in

*ctl->name2* along with any data posted with the event. The queue space and queue name can be any valid BEA Tuxedo ATMI system queue space and queue name, either of which may or may not exist at the time the subscription is made.

*ctl->qctl* can contain options further directing the EventBroker's enqueuing of the posted event. If no options are specified, then *ctl->qctl.flags* should be set to TPNOFLAGS. Otherwise, options can be set as described in the "Control Parameter" subsection of tpenqueue(3c) (specifically, see the section describing the valid list of flags controlling input information for tpenqueue(3c)). TPEVSERVICE and TPEVQUEUE are mutually exclusive flags.

If TPEVTRAN is also set in *ctl->flags*, then if the process calling tppost() is in transaction mode, the EventBroker enqueues the posted event and its data such that it will be part of the poster's transaction. The EventBroker, TMUSREVT(5), must belong to a server group that supports transactions (see UBBCONFIG(5) for details). If TPEVTRAN is not set in *ctl->flags*, then the EventBroker enqueues the posted event and its data such that it will not be part of the poster's transaction.

TPEVTRAN

Setting this flag indicates that the subscriber wants the event notification for this subscription to be included in the poster's transaction, if one exists. If the poster is not a transaction, then a transaction is started for this event notification. If this flag is not set, then any events posted for this subscription will not be done on behalf of any transaction in which the poster is participating. This flag can be used with either TPEVSERVICE or TPEVQUEUE.

TPEVPERSIST

By default, the BEA Tuxedo EventBroker deletes subscriptions when the resource to which it is posting is not available (for example, the EventBroker cannot access a service routine and/or a queue space/queue name associated with an event subscription). Setting this flag indicates that the subscriber wants this subscription to persist across such errors (usually because the resource will become available again in the future). When this flag is not used, the EventBroker will remove this subscription if it encounters an error accessing either the service name or queue space/queue name designated in this subscription.

If this flag is used with TPEVTRAN and the resource is not available at the time of event notification, then the EventBroker will return to the poster such that its transaction must be aborted. That is, even though the subscription remains intact, the resource's unavailability will cause the poster's transaction to fail.

If the EventBroker's list of active subscriptions already contains a subscription that matches the one being requested by tpsubscribe(), then the function fails setting tperrno to TPEMATCH. For a subscription to match an existing one, both *eventexpr* and *filter* must match those of a

subscription already in the EventBroker's active list of subscriptions. In addition, depending on the notification method, other criteria are used to determine matches.

If the subscriber is a BEA Tuxedo ATMI system client process and *ctl* is NULL (such that the caller receives unsolicited notifications when events are posted), then its system-defined client identifier (known as a CLIENTID) is also used to detect matches. That is, tpsubscribe() fails if *eventexpr*, *filter*, and the caller's CLIENTID match those of a subscription already known to the EventBroker.

If the caller has set *ctl→flags* to TPEVSERVICE, then tpsubscribe() fails if *eventexpr*, *filter*, and the service name set in *ctl→name1* match those of a subscription already known to the EventBroker.

For subscriptions to stable-storage queues, the queue space, queue name, and correlation identifier are used, in addition to *eventexpr* and *filter*, when determining matches. The correlation identifier can be used to differentiate among several subscriptions for the same event expression and filter rule, destined for the same queue. Thus, if the caller has set *ctl→flags* to TPEVQUEUE, and TPQCOORID is not set in *ctl→qctl.flags*, then tpsubscribe() fails if *eventexpr*, *filter*, the queue space name set in *ctl→name1*, and the queue name set in *ctl→name2* match those of a subscription (which also does not have a correlation identifier specified) already known to the EventBroker. Further, if TPQCOORID is set in *ctl→qctl.flags*, then tpsubscribe() fails if *eventexpr*, *filter*, *ctl→name1*, *ctl→name2*, and *ctl→qctl.corrid* match those of a subscription (which has the same correlation identifier specified) already known to the EventBroker.

The following is a list of valid *flags* for tpsubscribe():

TPNOBLOCK
: The subscription is not made if a blocking condition exists. If such a condition occurs, the call fails and tperrno is set to TPEBLOCK. When TPNOBLOCK is not specified and a blocking condition exists, the caller blocks until the condition subsides or a timeout occurs (either transaction or blocking timeout).

TPNOTIME
: This flag signifies that the caller is willing to block indefinitely and wants to be immune to blocking timeouts. Transaction timeouts may still occur.

TPSIGRSTRT
: If a signal interrupts any underlying system calls, then the interrupted system call is reissued. When TPSIGRSTRT is not specified and a signal interrupts a system call, then tpsubscribe() fails and tperrno is set to TPGOTSIG.

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tpsubscribe().

## Regular Expressions

The regular expressions described in Table 11 are much like those used in the UNIX system editor, ed(1). The alternation operator, (|), has been added along with some other practical things. In general, however, there should be few surprises.

Regular expressions (REs) are constructed by applying any of the following production rules one or more times.

**Table 11 Regular Expressions**

| Rule | Matching Text |
|---|---|
| character | Itself (*character* is any ASCII character except the special ones mentioned below). |
| \ character | Itself except as follows:<br>• \\—newline<br>• \\t—tab<br>• \\b—backspace<br>• \\r—carriage return<br>• \\f—formfeed |
| \ special-character | Its *un*special self. The special characters are . * + ? \| ( ) [ { and \\.<br>.—Any character except the end-of-line character (usually newline or NULL).<br>^—Beginning of the line.<br>$—End-of-line character. |
| [class] | any character in the class denoted by a sequence of characters and/or ranges. A range is given by the construct *character-character*. For example, the character class, [a-zA-Z0-9_], will match any alphameric character or "_". To be included in the class, a hyphen, "-", must be escaped (preceded by a "\\") or appear first or last in the class. A literal "]" must be escaped or appear first in the class. A literal "^" must be escaped if it appears first in the class. |
| [^ class ] | Any character in the complement of the class with respect to the ASCII character set, excluding the end-of-line character. |
| RE RE | The sequence. (catenation) |
| RE \| RE | Either the left RE or the right RE. (left to right alternation) |

**Table 11 Regular Expressions (Continued)**

| Rule | Matching Text |
|---|---|
| `RE *` | Zero or more occurrences of RE. |
| `RE +` | One or more occurrences of RE. |
| `RE ?` | Zero or one occurrences of RE. |
| `RE { n }` | *n* occurrences of RE. *n* must be between 0 and 255, inclusive. |
| `RE { m, n }` | `m` through `n` occurrences of RE, inclusive. A missing *m* is taken to be zero. A missing *n* denotes *m* or more occurrences of `RE`. |
| `( RE )` | Explicit precedence/grouping. |
| `( RE ) $ n` | The text matching `RE` is copied into the *n*th user buffer. *n* may be 0 through 9. User buffers are cleared before matching begins and loaded only if the entire pattern is matched. |

There are three levels of precedence. In order of decreasing binding strength they are:

- `catenation closure (*,+,?,{...})`
- `catenation`
- `alternation (|)`

As indicated above, parentheses are used to give explicit precedence.

## Return Values

Upon successful completion, `tpsubscribe()` returns a handle that can be used to remove this subscription from the EventBroker's list of active subscriptions. The subscriber or any other process is allowed to use the returned handle to delete this subscription.

Upon failure, `tpsubscribe()` returns -1 and sets `tperrno` to indicate the error condition.

## Errors

Upon failure, `tpsubscribe()` sets `tperrno` to one of the following values. (Unless otherwise noted, failure does not affect the caller's transaction, if one exists.)

**[**`TPEINVAL`**]**

Invalid arguments were given (for example, *`eventexpr`* is NULL).

[TPENOENT]
Cannot access the BEA Tuxedo EventBroker.

[TPELIMIT]
The subscription failed because the EventBroker's maximum number of subscriptions has been reached.

[TPEMATCH]
The subscription failed because it matched one already listed with the EventBroker.

[TPEPERM]
The client is not attached as tpsysadm and the subscription action is either a service call or the enqueuing of a message.

[TPETIME]
This error code indicates that either a timeout has occurred or tpsubscribe() has been attempted, in spite of the fact that the current transaction is already marked rollback only.

If the caller is in transaction mode, then either the transaction is already rollback only or a transaction timeout has occurred. The transaction is marked abort-only. If the caller is not in transaction mode, a blocking timeout has occurred. (A blocking timeout cannot occur if TPNOBLOCK and/or TPNOTIME is specified.)

If a transaction timeout has occurred, then, with one exception, any attempts to send new requests or receive outstanding replies will fail with TPETIME until the transaction has been aborted. The exception is a request that does not block, expects no reply, and is not sent on behalf of the caller's transaction (that is, tpacall() with TPNOTRAN, TPNOBLOCK, and TPNOREPLY set).

When a transactional ATMI call fails inside a transaction, the transaction is put into the TX_ROLLBACK_ONLY state. This state is treated, for most purposes, as though it were equivalent to a timeout. All further ATMI calls for this transaction (with the exception of those issued in the circumstances described in the previous paragraph) will fail with TPETIME.

[TPEBLOCK]
A blocking condition exists and TPNOBLOCK was specified.

[TPGOTSIG]
A signal was received and TPSIGRSTRT was not specified.

[TPEPROTO]
tpsubscribe() was called improperly.

**[**TPESYSTEM**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPEOS**]**
An operating system error has occurred.

### See Also

buffer(3c), tpenqueue(3c), tppost(3c), tpsetunsol(3c), tpunsubscribe(3c), Fboolco, Fboolco32, Fvboolco, Fvboolco32(3fml), Fboolev, Fboolev32, Fvboolev, Fvboolev32(3fml), EVENTS(5), EVENT_MIB(5), TMSYSEVT(5), TMUSREVT(5), tuxtypes(5), typesw(5), UBBCONFIG(5)

## tpsuspend(3c)

### Name

tpsuspend()—Suspends a global transaction.

### Synopsis

```
#include <atmi.h>
int tpsuspend(TPTRANID *tranid, long flags)
```

### Description

tpsuspend() is used to suspend the transaction active in the caller's process. A transaction begun with tpbegin() may be suspended with tpsuspend(). Either the suspending process or another process may use tpresume() to resume work on a suspended transaction. When tpsuspend() returns, the caller is no longer in transaction mode. However, while a transaction is suspended, all resources associated with that transaction (such as database locks) remain active. Like an active transaction, a suspended transaction is susceptible to the transaction timeout value that was assigned when the transaction first began.

For the transaction to be resumed in another process, the caller of tpsuspend() must have been the initiator of the transaction by explicitly calling tpbegin(). tpsuspend() may also be called by a process other than the originator of the transaction (for example, a server that receives a request in transaction mode). In the latter case, only the caller of tpsuspend() may call tpresume() to resume that transaction. This case is allowed so that a process can temporarily suspend a transaction to begin and do some work in another transaction before completing the

original transaction (for example, to run a transaction to log a failure before rolling back the original transaction).

tpsuspend() returns in the space pointed to by *tranid* the transaction identifier being suspended. The caller is responsible for allocating the space to which *tranid* points. It is an error for *tranid* to be NULL.

To ensure success, the caller must have completed all outstanding transactional communication with servers before issuing tpsuspend(). That is, the caller must have received all replies for requests sent with tpacall() that were associated with the caller's transaction. Also, the caller must have closed all connections with conversational services associated with the caller's transaction (that is, tprecv() must have returned the TPEV_SVCSUCC event). If either rule is not followed, then tpsuspend() fails, the caller's current transaction is not suspended and all transactional communication descriptors remain valid. Communication descriptors not associated with the caller's transaction remain valid regardless of the outcome of tpsuspend().

Currently, *flags* are reserved for future use and must be set to 0.

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tpsuspend().

## Return Value

tpsuspend() returns -1 on error and sets tperrno to indicate the error condition.

## Errors

Under the following conditions, tpsuspend() fails and sets tperrno to:

**[**TPEINVAL**]**
: *tranid* is a NULL pointer or *flags* is not 0. The caller's state with respect to the transaction is not changed.

**[**TPEABORT**]**
: The caller's active transaction has been aborted. All communication descriptors associated with the transaction are no longer valid.

**[**TPEPROTO**]**
: tpsuspend() was called in an improper context (for example, the caller is not in transaction mode). The caller's state with respect to the transaction is not changed.

**[**TPESYSTEM**]**
: A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPEOS**]**
An operating system error has occurred.

### See Also

tpacall(3c), tpbegin(3c), tprecv(3c), tpresume(3c)

## tpsvrdone(3c)

### Name

tpsvrdone()—Terminates a BEA Tuxedo ATMI system server.

### Synopsis

```
#include <atmi.h>
void tpsvrdone(void)
```

### Description

The BEA Tuxedo ATMI system server abstraction calls tpsvrdone() after it has finished processing service requests but before it exits. When this routine is invoked, the server is still part of the system but its own services have been unadvertised. Thus, BEA Tuxedo ATMI system communication can be performed and transactions can be defined in this routine. However, if tpsvrdone() returns with open connections, asynchronous replies pending or while still in transaction mode, the BEA Tuxedo ATMI system will close its connections, ignore any pending replies, and abort the transaction before the server exits.

If a server is shut down by the invocation of tmshutdown -y, services are suspended and the ability to perform communication or to begin transactions in tpsvrdone() is limited.

If an application does not provide this routine in a server, then the default version provided by the BEA Tuxedo ATMI system is called instead. If a server has been defined as a single-threaded server, the default tpsvrdone() calls tpsvrthrdone(), and the default version of tpsvrthrdone() calls tx_close(). If a server has been defined as a multithreaded server, tpsvrthrdone() is called in each server dispatch thread, but is not called from tpsvrdone(). Regardless of whether the server is multithreaded, the default tpsvrdone() calls userlog to indicate that the server is about to exit.

### Usage

When called in tpsvrdone(), the tpreturn() and tpforward() functions simply return with no effect.

### See Also

`tpsvrthrdone(3c)`, `tpsvrthrinit(3c)`, `servopts(5)`

## tpsvrinit(3c)

### Name

`tpsvrinit()`—Initializes a BEA Tuxedo system server.

### Synopsis

```
#include <atmi.h>
int tpsvrinit(int argc, char **argv)
```

### Description

The BEA Tuxedo ATMI system server abstraction calls `tpsvrinit()` during its initialization. This routine is called after the thread of control has become a server but before it handles any service requests; thus, BEA Tuxedo ATMI system communication may be performed and transactions may be defined in this routine. However, if `tpsvrinit()` returns with either open connections or asynchronous replies pending, or while still in transaction mode, the BEA Tuxedo ATMI system closes the connections, ignores any pending replies, and aborts the transaction before the server exits.

If an application does not provide this routine in a server, then the default version provided by the BEA Tuxedo ATMI system is called, instead.

If a server has been defined as a single-threaded server, the default `tpsvrinit()` calls `tpsvrthrinit()`, and the default version of `tpsvrthrinit()` calls `tx_open()`. If a server has been defined as a multithreaded server, `tpsvrthrinit()` is called in each server dispatch thread, but is not called from `tpsvrinit()`. Regardless of whether the server is single-threaded or multithreaded, the default version of `tpsvrinit()` calls `userlog()` to indicate that the server started successfully.

Application-specific options can be passed into a server and processed in `tpsvrinit()` (see `servopts(5)`). The options are passed through *`argc`* and *`argv`*. Since `getopt()` is used in a BEA Tuxedo ATMI system server abstraction, `optarg()`, `optind()`, and `opterr()` may be used to control option parsing and error detection in `tpsvrinit()`.

**Note:** When invoking `tpsvrinit()` in your code, avoid long blocking actions. Otherwise, when one remote server in an MP configuration has trouble with `tpsvrinit()` processing, then `tmboot` fails to boot the other servers on that node.

If an error occurs in `tpsvrinit()`, the application can cause the server to exit gracefully (and not take any service requests) by returning -1. The application itself should not call `exit()`.

When `tpsvrinit()` returns -1, the system does not restart the server. Instead, the administrator must run `tmboot` to restart the server.

### Return Values

A negative return value causes the server to exit gracefully.

### Usage

When used outside a service routine (for example, in clients, in `tpsvrinit()`, or in `tpsvrdone()`), the `tpreturn()` and `tpforward()` functions simply return with no effect.

### See Also

`tpopen(3c)`, `tpsvrdone(3c)`, `tpsvrthrinit(3c)`, `servopts(5)`
`getopt`(3) in a C language reference manual

## tpsvrthrdone(3c)

### Name

`tpsvrthrdone()`—Terminates a BEA Tuxedo ATMI server thread.

### Synopsis

```
#include <atmi.h>
void tpsvrthrdone(void)
```

### Description

The BEA Tuxedo ATMI server abstraction calls `tpsvrthrdone()` during the termination of each thread that has been started to handle dispatched service requests. In other words, even if a thread is terminated before it has handled a request, the `tpsvrdone()` function is called. When this routine is called, the thread of control is still part of the BEA Tuxedo ATMI server, but the thread has finished processing all service requests. Thus, BEA Tuxedo ATMI communication may be performed and transactions may be defined in this routine. However, if `tpsvrthrdone()` returns with either open connections or asynchronous replies pending, or while still in transaction mode, the BEA Tuxedo ATMI system closes the connections, ignores any pending replies, and aborts the transaction before the server dispatch thread exits.

If an application does not provide this routine in a server, then the default version of `tpsvrthrdone()` provided by the BEA Tuxedo ATMI system is called instead. The default version of `tpsvrthrdone()` calls `tx_close()`.

`tpsvrthrdone()` is called even in single-threaded servers. In a single-threaded server, `tpsvrthrdone()` is called from the default version of `tpsvrdone()`. In a server with the potential for multiple dispatch threads, `tpsvrdone()` does not call `tpsvrthrdone()`.

### Usage

When called from `tpsvrthrdone()`, the `tpreturn()` and `tpforward()` functions simply return with no effect.

### See Also

`tpforward(3c)`, `tpreturn(3c)`, `tpsvrdone(3c)`, `tpsvrthrinit(3c)`, `tx_close(3c)`, `servopts(5)`

## tpsvrthrinit(3c)

### Name

`tpsvrthrinit()`—Initializes a BEA Tuxedo ATMI server thread.

### Synopsis

```
#include <atmi.h>
int tpsvrthrinit(int argc, char **argv)
```

### Description

The BEA Tuxedo ATMI server abstraction calls `tpsvrthrinit()` during the initialization of each thread that handles dispatched service requests. This routine is called after the thread of control has become part of the BEA Tuxedo ATMI server but before the thread handles any service requests. Thus, BEA Tuxedo ATMI communication may be performed and transactions may be defined in this routine. However, if `tpsvrthrinit()` returns with either open connections or asynchronous replies pending, or while still in transaction mode, the BEA Tuxedo ATMI system closes the connections, ignores any pending replies, and aborts the transaction before the server dispatch thread exits.

If an application does not provide this routine in a server, then the default version of `tpsvrthrinit()` provided by the BEA Tuxedo ATMI system is called instead. The default version of `tpsvrthrinit()` calls `tx_open()`.

`tpsvrthrinit()` is called even in single-threaded servers. In a single-threaded server, `tpsvrthrinit()` is called from the default version of `tpsvrinit()`. In a server with the potential for multiple dispatch threads, `tpsvrinit()` does not call `tpsvrthrinit()`.

Application-specific options can be passed into a server and processed in `tpsvrthrinit()`. For more information about options, see `servopts(5)`. The options are passed *`argc`* and *`argv`*. Because `getopt()` is used in a BEA Tuxedo ATMI server abstraction, `optarg()`, `optind()`, and `opterr()` may be used to control option parsing and error detection in `tpsvrthrinit()`.

If an error occurs in `tpsvrthrinit()`, the application can cause the server dispatch thread to exit gracefully (and not take any service requests) by returning -1. The application should not call `exit()` or any operating system thread exit function.

### Return Values

A negative return value will cause the server dispatch thread to exit gracefully.

### Usage

When used outside a service routine (for example, when used in a client or in `tpsvrinit()`, `tpsvrdone()`, `tpsvrthrinit()`, or `tpsvrthrdone()`), the `tpreturn()` and `tpforward()` functions simply return with no effect.

### See Also

`tpforward(3c)`, `tpreturn(3c)`, `tpsvrthrdone(3c)`, `tpsvrthrinit(3c)`, `tx_open(3c)`, `servopts(5)`

`getopt`(3) in a C language reference manual

## tpterm(3c)

### Name

`tpterm()`—Leaves an application.

### Synopsis

```
#include <atmi.h>
int tpterm(void)
```

### Description

`tpterm()` removes a client from a BEA Tuxedo ATMI system application. If the client is in transaction mode, then the transaction is rolled back. When `tpterm()` returns successfully, the

caller can no longer perform BEA Tuxedo ATMI client operations. Any outstanding conversations are immediately disconnected.

If tpterm() is called more than once (that is, if it is called after the caller has already left the application), no action is taken and success is returned.

## Multithreading and Multicontexting Issues

In good programming practice, all threads but one should either exit or switch context before the single remaining thread issues a call to tpterm(). If this is not done, then the remaining threads are put in a TPINVALIDCONTEXT context. A description of the semantics of this context follows.

When invoked by one thread in a context with which multiple threads are associated, tpterm():

- Operates on all threads in a context, but not on all contexts in a process
- Executes immediately, even if other threads in the same process are still associated with that context

Any thread blocked in an ATMI call when another thread terminates its context will return from the ATMI call with a failure return; tperrno is set to TPESYSTEM. In addition, if tperrordetail() is invoked after such a failure return, it returns TPED_INVALIDCONTEXT.

In a single-context application, whenever a single thread calls tpterm(), the context state for all threads is set to TPNULLCONTEXT.

In a multicontexted application, however, when tpterm() is invoked by one thread, all other threads in the same context are placed in a state such that if they subsequently call most ATMI functions, those functions will, instead, return failure with tperrno set to TPEPROTO. Lists of the functions that are allowed and disallowed in such an invalid context state are provided in “Introduction to the C Language Application-to-Transaction Monitor Interface” on page 8. If a thread in the invalid context state (TPINVALIDCONTEXT) calls the tpgetctxt() function, tpgetctxt() sets the context parameter to TPINVALIDCONTEXT.

A thread may exit from the TPINVALIDCONTEXT state by calling one of the following:

- tpsetctxt() with the TPNULLCONTEXT context or another valid context
- tpterm()

It is forbidden to call tpsetctxt() with a context of TPINVALIDCONTEXT; doing so results in failure with tperrno set to TPEPROTO. When a thread invokes ATMI functions other than tpsetunsol() that do not require the caller to be associated with an application, these functions behave as if they were invoked in the NULL context. Client applications using unsolicited thread notification should explicitly call tpterm() to terminate the unsolicited notification thread.

After invoking tpterm(), a thread is placed in the TPNULLCONTEXT context. Most ATMI functions invoked by a thread in the TPNULLCONTEXT context perform an implicit tpinit(). Whether or not the call to tpinit() succeeds depends on the usual determining factors, unrelated to context-specific or thread-specific issues.

A thread in a multithreaded application may issue a call to tpterm() while running in any context state, including TPINVALIDCONTEXT.

### Return Values

Upon success in a single-context application, all threads in the application's current context are placed in the TPNULLCONTEXT state.

Upon success in a multicontexted application, the calling thread is placed in the TPNULLCONTEXT state and all other threads in the same context as the calling thread are placed in the TPINVALIDCONTEXT state. The user may change the context state of the latter threads by running tpsetctxt() with the *context* argument set to TPNULLCONTEXT or another valid context.

Upon failure, tpterm() leaves the calling process in its original context state, returns -1, and sets tperrno to indicate the error condition.

### Errors

Upon failure, tpterm() sets tperrno to one of the following values:

**[**TPEPROTO**]**
: tpterm() was called in an improper context (for example, the caller is a server).

**[**TPESYSTEM**]**
: A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPEOS**]**
: An operating system error has occurred.

### See Also

tpinit(3c), tpgetctxt(3c), tpsetctxt(3c), tpsetunsol(3c)

## tptypes(3c)

### Name

tptypes()—Routine to determine information about a typed buffer.

## Synopsis

```
#include <atmi.h>
long tptypes(char *ptr, char *type, char *subtype)
```

## Description

`tptypes()` takes as its first argument a pointer to a data buffer and returns the type and subtype of that buffer in its second and third arguments, respectively. *`ptr`* must point to a buffer gotten from `tpalloc()`. If *`type`* and *`subtype`* are non-NULL, then the function populates the character arrays to which they point with the names of the buffer's type and subtype, respectively. If the names are of their maximum length (8 for *`type`*, 16 for *`subtype`*), the character array is not NULL-terminated. If no subtype exists, then the array pointed to by *`subtype`* will contain a NULL string.

Note that only the first eight bytes of *`type`* and the first 16 bytes of *`subtype`* are populated.

A thread in a multithreaded application may issue a call to `tptypes()` while running in any context state, including `TPINVALIDCONTEXT`.

## Return Values

Upon success, `tptypes()` returns the size of the buffer;

Upon failure, it returns -1 and sets `tperrno` to indicate the error condition.

## Errors

Upon failure, `tptypes()` sets tperrno to one of the following values:

**[**`TPEINVAL`**]**
: Invalid arguments were given (for example, *`ptr`* does not point to a buffer gotten from `\% tpalloc()`).

**[**`TPEPROTO`**]**
: `tptypes()` was called improperly.

**[**`TPESYSTEM`**]**
: A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**`TPEOS`**]**
: An operating system error has occurred.

## See Also

`tpalloc(3c)`, `tpfree(3c)`, `tprealloc(3c)`

# tpunadvertise(3c)

## Name

`tpunadvertise()`—Routine for unadvertising a service name.

## Synopsis

```
#include <atmi.h>
int tpunadvertise(char *svcname)
```

## Description

`tpunadvertise()` allows a server to unadvertise a service that it offers. By default, a server's services are advertised when it is booted and they are unadvertised when it is shut down.

All servers belonging to a Multiple Server, Single Queue (MSSQ) set must offer the same set of services. These routines enforce this rule by affecting the advertisements of all servers sharing an MSSQ set.

`tpunadvertise()` removes *`svcname`* as an advertised service for the server (or the set of servers sharing the caller's MSSQ set). *`svcname`* cannot be NULL or the NULL string (""). Also, *`svcname`* should be 15 characters or less. (See the *SERVICES section of `UBBCONFIG(5)`). Longer names will be accepted and truncated to 15 characters. Care should be taken such that truncated names do not match other service names.

## Return Values

Upon failure, `tpunadvertise()` returns -1 and sets `tperrno` to indicate the error condition.

## Errors

Upon failure, `tpunadvertise()` sets `tperrno` to one of the following values:

**[**`TPEINVAL`**]**
*`svcname`* is NULL or the NULL string ("").

**[**`TPENOENT`**]**
*`svcname`* is not currently advertised by the server.

**[**`TPEPROTO`**]**
`tpunadvertise()` was called in an improper context (for example, by a client).

**[**`TPESYSTEM`**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

[TPEOS]
An operating system error has occurred.

### See Also

tpadvertise(3c)

## tpunsubscribe(3c)

### Name

tpunsubscribe()—Unsubscribes to an event.

### Synopsis

```
#include <atmi.h>
int tpunsubscribe(long subscription, long flags)
```

### Description

The caller uses tpunsubscribe() to remove an event subscription or a set of event subscriptions from the BEA Tuxedo EventBroker's list of active subscriptions. *subscription* is an event subscription handle returned by tpsubscribe(). Setting *subscription* to the wildcard value, -1, directs tpunsubscribe() to unsubscribe to all non-persistent subscriptions previously made by the calling process. Non-persistent subscriptions are those made without the TPEVPERSIST bit setting in the *ctl*≫*flags* parameter of tpsubscribe(). Persistent subscriptions can be deleted only by using the handle returned by tpsubscribe().

Note that the -1 handle removes only those subscriptions made by the calling process and not any made by previous instantiations of the caller (for example, a server that dies and restarts cannot use the wildcard to unsubscribe to any subscriptions made by the original server).

The following is a list of valid *flags*:

TPNOBLOCK
: The subscription is not removed if a blocking condition exists. If such a condition occurs, the call fails and tperrno is set to TPEBLOCK. When TPNOBLOCK is not specified and a blocking condition exists, the caller blocks until the condition subsides or a timeout occurs (either transaction or blocking timeout).

TPNOTIME
: This flag signifies that the caller is willing to block indefinitely and wants to be immune to blocking timeouts. Transaction timeouts may still occur.

TPSIGRSTRT
: If a signal interrupts any underlying system calls, then the interrupted system call is reissued. When TPSIGRSTRT is not specified and a signal interrupts a system call, then tpunsubscribe() fails and tperrno is set to TPGOTSIG.

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tpunsubscribe().

## Return Values

Upon completion of tpunsubscribe(), tpurcode() contains the number of subscriptions deleted (zero or greater) from the EventBroker's list of active subscriptions. tpurcode() may contain a number greater than 1 only when the wildcard handle, -1, is used. Also, tpurcode() may contain a number greater than 0 even when tpunsubscribe() completes unsuccessfully (that is, when the wildcard handle is used, the EventBroker may have successfully removed some subscriptions before it encountered an error deleting others).

Upon failure, tpunsubscribe() returns -1 and sets tperrno to indicate the error condition.

## Errors

Upon failure, tpunsubscribe() sets tperrno to one of the following values. (Unless otherwise noted, failure does not affect the caller's transaction, if one exists.)

**[**TPEINVAL**]**
: Invalid arguments were given (for example, *subscription* is an invalid subscription handle).

**[**TPENOENT**]**
: Cannot access the BEA Tuxedo EventBroker.

**[**TPETIME**]**
: This error code indicates that either a timeout has occurred or tpunsubscribe() has been attempted, in spite of the fact that the current transaction is already marked rollback only.

  If the caller is in transaction mode, then either the transaction is already rollback only or a transaction timeout has occurred. The transaction is marked abort-only. If the caller is not in transaction mode, a blocking timeout has occurred. (A blocking timeout cannot occur if TPNOBLOCK and/or TPNOTIME is specified.)

  If a transaction timeout has occurred, then, with one exception, any attempts to send new requests or receive outstanding replies will fail with TPETIME until the transaction has been aborted. The exception is a request that does not block, expects no reply, and is not

sent on behalf of the caller's transaction (that is, tpacall() with TPNOTRAN, TPNOBLOCK, and TPNOREPLY set).

When a transactional ATMI call fails inside a transaction, the transaction is put into the TX_ROLLBACK_ONLY state. This state is treated, for most purposes, as though it were equivalent to a timeout. All further ATMI calls for this transaction (with the exception of those issued in the circumstances described in the previous paragraph) will fail with TPETIME.

**[**TPEBLOCK**]**
A blocking condition exists and TPNOBLOCK was specified.

**[**TPGOTSIG**]**
A signal was received and TPSIGRSTRT was not specified.

**[**TPEPROTO**]**
tpunsubscribe() was called improperly.

**[**TPESYSTEM**]**
A BEA Tuxedo system error has occurred. The exact nature of the error is written to a log file.

**[**TPEOS**]**
An operating system error has occurred.

### See Also

tppost(3c), tpsubscribe(3c), EVENTS(5), EVENT_MIB(5), TMSYSEVT(5), TMUSREVT(5)

## tputrace(3c)

### Name

tputrace()—User-defined trace information application.

### Synopsis

```
#include <atmi.h>
int tputrace (char *trrec, int nest, char *category, char *funcname, int
utrtype, va_list args)
```

### Description

tputrace(3c) is a *user-defined* API the allows flexibility in monitoring and obtaining detailed trace output information (such as full user data content that is passed to or returned from ATMI

functions) and defines how and where this information is output. By default, tputrace() outputs trace record information to userlog(3c) if the user does not update or modify otherwise.

tputrace(3c) is called exclusively by specifying the utrace receiver with TMTRACE. For example: TMTRACE=atmi:utrace. Specifying the utrace receiver automatically invokes tputrace(3c) and applies it only to atmi trace category records for output. For more TMTRACE and utrace receiver information, see tmtrace(5) in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*.

Valid tputrace(3c) arguments are:

trrec
: Trace information record defined by user. ttrec is *always* specified as the first argument in tputrace(3c). Outputting trrec to the userlog, produces the same results as tmtrace(5).

nest
: Defines the nesting level. Use this if indentations are added to the tputrace(3c) output lines.

category
: Defines the ATMI function category, for example "atmi", "iatmi" or "xa".

funcname
: Defines the function name. For example, "tpcall" or "tconnect".

utrtype
: Indicates whether tputrace(3c) is called when entering or leaving an ATMI function. Set values as follows: 0 = entering, 1 = leaving.

args
: Define arguments passed to the tputrace(3c) output function. This includes user data or flags passed to ATMI functions. The list of the arguments for each ATMI functions are defined in the tputrace() example implementation in the Example(s) section. The argument list is also available from tmtrace(5) trace information output.

## Libutrace Library

A separate Tuxedo library, libutrace, is used in conjunction with tputrace(). The default libutrace is installed in the Tuxedo system shared library directory ($TUXDIR/lib in UNIX and %TUXDIR%\bin in Windows).

Users can also write their own their own custom libutrace library and can install it in either:

- the Tuxedo system shared library directory, or
- the Tuxedo application directory ($APPDIR in UNIX and %APPDIR% in Windows).

If the custom `libutrace` library is installed in the system directory, it replaces the default `libutrace` and is used by all Tuxedo clients and servers on the machine. If the custom `libutrace` library is installed in the application directory, it is used only by the clients and the servers in the application.

Whenever `tputrace()` is modified, the `libutrace` library *must* be recompiled and linked to Tuxedo 9.0 or later. A sample `tputrace()` source file is located in the `$TUXDIR/samples/atmi/libutrace` directory.

The Example(s) section further illustrates how to customize `tputrace()`.

**Warning:** The default or custom `libutrace` library is loaded into *every* Tuxedo application process, including system servers such as BBL or WSL. This being the case, all Tuxedo system servers consume some amount of memory for loading `libutrace`. The *default* `libutrace` library is very small so memory consumption is negligible. But a *custom* libutrace can consume a larger amount of memory depending on how much functionality the user adds.

## Example(s)

This is example shows the user-level trace information userlog output for the `simpcl` execution of the Tuxedo `simpapp` sample program.

In order to customize user-level trace information and output, you must do the following:

1. Modify `tputrace()`.
2. Re-compile the `libutrace` library and link to Tuxedo.

For this example, when `TMTRACE=atmi:utrace` is specified it writes the contents of the user data and flags passed to the ATMI functions to the Tuxedo userlog.

**Listing 1 Simpapp Sample User-Level Trace Information Userlog Output**

```
091206.HOST1!?proc.1560.1520.0: UTRAC:at:  } tpinit = 1
091206.HOST1!?proc.1560.1520.0: UTRAC:at:  { tpalloc("STRING", "", 7)
091206.HOST1!?proc.1560.1520.0: UTRAC:at:  } tpalloc = 0x86a8e8
091206.HOST1!?proc.1560.1520.0: UTRAC:at:  { tpalloc("STRING", "", 7)
091206.HOST1!?proc.1560.1520.0: UTRAC:at:  } tpalloc = 0x87fa20
091206.HOST1!?proc.1560.1520.0: UTRAC:at:  { tpcall(
091206.HOST1!?proc.1560.1520.0: UTRAC:at:      svc="TOUPPER"
091206.HOST1!?proc.1560.1520.0: UTRAC:at:      idata=(0x86a8e8){
```

```
091206.HOST1!?proc.1560.1520.0: UTRAC:at:        len=0
091206.HOST1!?proc.1560.1520.0: UTRAC:at:        type="STRING"
091206.HOST1!?proc.1560.1520.0: UTRAC:at:        value="abcdef"
091206.HOST1!?proc.1560.1520.0: UTRAC:at:      }
091206.HOST1!?proc.1560.1520.0: UTRAC:at:      odata=(0x12ff48){
091206.HOST1!?proc.1560.1520.0: UTRAC:at:        data=(0x87fa20){
091206.HOST1!?proc.1560.1520.0: UTRAC:at:          len=0
091206.HOST1!?proc.1560.1520.0: UTRAC:at:          type="STRING"
091207.HOST1!?proc.1560.1520.0: UTRAC:at:        }
091207.HOST1!?proc.1560.1520.0: UTRAC:at:        len=(0x12ff44)0
091207.HOST1!?proc.1560.1520.0: UTRAC:at:      }
091207.HOST1!?proc.1560.1520.0: UTRAC:at:      flags=<none>
091207.HOST1!?proc.1560.1520.0: UTRAC:at:    )
091207.HOST1!simpserv.760.2188.0: UTRAC:at:  { tpservice(
091207.HOST1!simpserv.760.2188.0: UTRAC:at:      svcinfo=(0x5e1518){
091207.HOST1!simpserv.760.2188.0: UTRAC:at:        name="TOUPPER"
091207.HOST1!simpserv.760.2188.0: UTRAC:at:        flags=<none>
091207.HOST1!simpserv.760.2188.0: UTRAC:at:        data=(0x602820){
091207.HOST1!simpserv.760.2188.0: UTRAC:at:          len=7
091207.HOST1!simpserv.760.2188.0: UTRAC:at:          type="STRING"
091207.HOST1!simpserv.760.2188.0: UTRAC:at:          value="abcdef"
091207.HOST1!simpserv.760.2188.0: UTRAC:at:        }
091207.HOST1!simpserv.760.2188.0: UTRAC:at:        cd=0
091207.HOST1!simpserv.760.2188.0: UTRAC:at:        appkey=0
091207.HOST1!simpserv.760.2188.0: UTRAC:at:
cltid=(0x5e154c){1095811926,0,12,0}
091207.HOST1!simpserv.760.2188.0: UTRAC:at:      }
091207.HOST1!simpserv.760.2188.0: UTRAC:at:    )
091207.HOST1!simpserv.760.2188.0: UTRAC:at:    { tpreturn(
091207.HOST1!simpserv.760.2188.0: UTRAC:at:        rval=TPSUCCESS
091207.HOST1!simpserv.760.2188.0: UTRAC:at:        rcode=0
091207.HOST1!simpserv.760.2188.0: UTRAC:at:        data=(0x602820){
091207.HOST1!simpserv.760.2188.0: UTRAC:at:          len=0
091207.HOST1!simpserv.760.2188.0: UTRAC:at:          type="STRING"
091207.HOST1!simpserv.760.2188.0: UTRAC:at:          value="ABCDEF"
091207.HOST1!simpserv.760.2188.0: UTRAC:at:        }
091207.HOST1!simpserv.760.2188.0: UTRAC:at:        flags=<none>
091207.HOST1!simpserv.760.2188.0: UTRAC:at:      )
```

```
091207.HOST1!?proc.1560.1520.0: UTRAC:at:  } tpcall(
091207.HOST1!?proc.1560.1520.0: UTRAC:at:      ret=0
091207.HOST1!?proc.1560.1520.0: UTRAC:at:      odata=(0x12ff48){
091207.HOST1!?proc.1560.1520.0: UTRAC:at:        data=(0x881690){
091207.HOST1!?proc.1560.1520.0: UTRAC:at:          len=7
|091207.HOST1!?proc.1560.1520.0: UTRAC:at:           type="STRING"
091207.HOST1!?proc.1560.1520.0: UTRAC:at:          value="ABCDEF"
091207.HOST1!?proc.1560.1520.0: UTRAC:at:        }
091207.HOST1!?proc.1560.1520.0: UTRAC:at:        len=(0x12ff44)7
091207.HOST1!simpserv.760.2188.0: UTRAC:at:    } tpreturn [long jump]
091207.HOST1!?proc.1560.1520.0: UTRAC:at:      }
091207.HOST1!?proc.1560.1520.0: UTRAC:at:    )
091207.HOST1!?proc.1560.1520.0: UTRAC:at:  { tpfree(0x86a8e8)
091207.HOST1!?proc.1560.1520.0: UTRAC:at:  } tpfree
091207.HOST1!?proc.1560.1520.0: UTRAC:at:  { tpfree(0x881690)
091207.HOST1!?proc.1560.1520.0: UTRAC:at:  } tpfree
091207.HOST1!simpserv.760.2188.0: UTRAC:at:  } tpservice
091207.HOST1!?proc.1560.1520.0: UTRAC:at:  { tpterm()
091207.HOST1!?proc.1560.1520.0: UTRAC:at:  } tpterm = 1
091207.HOST1!?proc.1560.1520.-2: UTRAC:at:  { tpterm()
091207.HOST1!?proc.1560.1520.-2: UTRAC:at:  } tpterm = 1
```

## Return Values

`tmutrace(3c)` returns `0` when run successfully, and return `-1` when a failure occurs.

## Errors

Failure depends on the `tputrace()` user-level implementation/customization. The default `tputrace()` implementation included in Tuxedo 9.0 or later does not cause failure.

## See Also

- `tmtrace(5)`
- `userlog(3c)`
- *Using the Run-time and User-level tracing utilities* in *Monitoring Your BEA Tuxedo Application*

# tpxmltofml32(3c)

## Name

`tpxmltofml32()`—Converts XML data to FML32 buffers

## Synopsis

```
#include <fml32.h>
int tpxmltofml32 (char *xmlbufp, char *vfile, FBFR32 **fml32bufp, char
**rtag, long flags)
```

## Description

This function is used to convert XML buffers to FML32 buffers. It supports the following valid arguments:

`xmlbufp`
: This argument is a pointer to valid XML typed buffer input.

`vfile`
: The argument is the fully qualified path name of an XML Schema file used to validate XML input. To use this argument, you must set the `TPXPARSNSPACE` and `TPXPARSDOSCHE` flags and *not* set the `TPXPARSNEVER` flag.

`fml32bufp`
: This argument is a pointer to an output FML32 typed buffer created from the input XML.

`rtag`
: This argument is a pointer that stores the root element name from the input XML document.

`flags`
: This argument is used in XML to FML/FML32 conversion to map to a Tuxedo 9.x subset of Xerces parser classes (see, XercesParser 2.5 documentation). The following is a list of Tuxedo 9.x valid Xerces parser `flags`:

  `TPXPARSNEVER`
  : Sets `setValidationScheme` to `Val_Never`. The parser will not report Schema validation errors.

  `TPXPARSALWAYS`
  : Sets `setValidationScheme` to `Val_Always`. The parser will always report Schema validation errors.

  **Note:** `TPXPARSNEVER` takes precedent over `TPXPARSALWAYS` if both arguments are used at the same time.

TPXPARSSCHFULL

Sets setValidationSchemaFullChecking to true. This flag allows the user to turn full Schema constraint checking on/off. Only takes effect if Schema validation is enabled. If turned off, partial constraint checking is done. Full schema constraint checking includes those checking that may be time-consuming or memory intensive. Currently, particle unique attribution constraint checking and particle derivation restriction checking are controlled by this option.

TPXPARSCONFATAL

Sets setValidationConstraintFatal to true. This flag allows users to set the parser's behavior when it encounters a validation constraint error. If set to true, and the parser will treat validation error as fatal and will exit depends on the state of getExitOnFirstFatalError. If false, then it will report the error and continue processing.

TPXPARSNSPACE

Sets setDoNamespaces to true. This flag allows users to enable or disable the parser's namespace processing. When set to true, parser starts enforcing all the constraints and rules specified by the NameSpace specification.

TPXPARSDOSCH

Sets setDoSchema to true. This flag allows users to enable or disable the parser's schema processing. When set to false, parser will not process any schema found.

**Note:** If set to true, namespace processing must also be turned on.

TPXPARSEREFN

Sets setCreateEntityReferencNodes to false. This flag allows the user to specify whether the parser should create entity reference nodes in the DOM tree being produced. When the *create* flag is true, the parser will create EntityReference nodes in the DOM tree. The EntityReference nodes and their child nodes will be read-only. When the *create* flag is false, no EntityReference nodes will be created.The replacement text of the entity is included in either case, either as a child of the Entity Reference node or in place at the location of the reference.

TPXPARSNOEXIT

Sets setExitOnFirstFatalError to false. This flag allows users to set the parser's behavior when it encounters the first fatal error. If set to true, the parser will exit at the first fatal error. If false, then it will report the error and continue processing.

TPXPARSNOINCWS

Sets setIncludeIgnorableWhitespace to false. This flag allows the user to specify whether a validating parser should include ignorable whitespaces as text nodes. It has no effect on non-validating parsers which always include non-markup text.

When set to false, all ignorable whitespace will be discarded and no text node is added to the DOM tree.

`TPXPARSCACHESET`
Sets `setcacheGrammarFromParse` to true. This flag allows users to enable or disable caching of grammar when parsing XML documents. When set to true, the parser will cache the resulting grammar for use in subsequent parses. If the flag is set to true, the *Use cached grammar* flag will also be set to true.

`TPXPARSCACHERESET`
Resets `resetCachedGrammarPool`. Resets the documents vector pool and release all the associated memory back to the system.

When parsing a document using a DOM parser, all memory allocated for a DOM tree is associated to the DOM document.

If you do multiple parse using the same DOM parser instance, then multiple DOM documents will be generated and saved in a vector pool. All these documents (and thus all the allocated memory) won't be deleted until the parser instance is destroyed.

If you do not need these DOM documents anymore and do not want to destroy the DOM parser instance at this moment, then you can call this method to reset the document vector pool and release all the allocated memory back to the system. It is an error to call this method if you are in the middle of a parse (for example, a mid progressive parse).

## Return Values

Upon success, `tpxmltofml32 ()` returns `0`. This function returns `-1` on error and sets `tperrno` to indicate the error condition.

## Errors

The function may fail for the following reasons:

**[**`TPEINVAL`**]**
Either `fml32bufp` or `xmlbufp` is not a valid typed buffer, or parser has problems understanding the input.

**[**`TPESYSTEM`**]**
A Tuxedo system error has occurred. The exact nature of the error is written to `userlog(3)`. This will also indicate when a conversion to FML32 was unable to be done. In that instance error detail info will be added to the userlog.

[TPEOS]
An operating system error has occurred. A numeric value representing the system call that failed is available in Uunixerr.

### SEE ALSO

tpfml32toxml(3c), tpxmltofml(3c),tpfmltoxml(3c)

## tpxmltofml(3c)

### Name

tpxmltofml()—Converts XML data to FML buffers

### Synopsis

```
#include <fml.h>
int tpxmltofml (char *xmlbufp, char *vfile, FBFR **fmlbufp, char **rtag,
long flags)
```

### Description

This function is used to convert XML data to FML buffers. It supports the following valid arguments:

xmlbufp
This argument is a pointer to valid XML typed buffer input.

vfile
The argument is the fully qualified path name of an XML Schema file used to validate XML input. To use this argument, you must set the TPXPARSNSPACE and TPXPARSDOSCHE flags and *not* set the TPXPARSNEVER flag.

fmlbufp
This argument is a pointer to an output FML typed buffer created from the input XML.

rtag
This argument is a pointer that stores the root element name from the input XML document.

flags
This argument is used in XML to FML/FM32L conversion to map to a Tuxedo 9.x subset of Xerces parser classes (see, XercesParser 2.5 documentation). The following is a list of Tuxedo 9.x valid Xerces parser flags:

TPXPARSNEVER
: Sets `setValidationScheme` to `Val_Never`. The parser will not report Schema validation errors.

TPXPARSALWAYS
: Sets `setValidationScheme` to `Val_Always`. The parser will always report Schema validation errors.

**Note:** `TPXPARSNEVER` takes precedent over `TPXPARSALWAYS` if both arguments are used at the same time.

TPXPARSSCHFULL
: Sets `setValidationSchemaFullChecking` to true. This flag allows the user to turn full Schema constraint checking on/off. Only takes effect if Schema validation is enabled. If turned off, partial constraint checking is done. Full schema constraint checking includes those checking that may be time-consuming or memory intensive. Currently, particle unique attribution constraint checking and particle derivation restriction checking are controlled by this option.

TPXPARSCONFATAL
: Sets `setValidationConstraintFatal` to true. This flag allows users to set the parser's behavior when it encounters a validation constraint error. If set to true, and the parser will treat validation error as fatal and will exit depends on the state of `getExitOnFirstFatalError`. If false, then it will report the error and continue processing.

TPXPARSNSPACE
: Sets `setDoNamespaces` to true. This flag allows users to enable or disable the parser's namespace processing. When set to true, parser starts enforcing all the constraints and rules specified by the `NameSpace` specification.

TPXPARSDOSCH
: Sets `setDoSchema` to true. This flag allows users to enable or disable the parser's schema processing. When set to false, parser will not process any schema found.

**Note:** If set to true, namespace processing must also be turned on.

TPXPARSEREFN
: Sets `setCreateEntityReferencNodes` to false. This flag allows the user to specify whether the parser should create entity reference nodes in the DOM tree being produced. When the *create* flag is true, the parser will create EntityReference nodes in the DOM tree. The EntityReference nodes and their child nodes will be read-only. When the *create* flag is false, no EntityReference nodes will be created.The replacement text of the entity is included in either case, either as a child of the Entity Reference node or in place at the location of the reference.

TPXPARSNOEXIT

Sets `setExitOnFirstFatalError` to false. This flag allows users to set the parser's behavior when it encounters the first fatal error. If set to true, the parser will exit at the first fatal error. If false, then it will report the error and continue processing.

TPXPARSNOINCWS

Sets `setIncludeIgnorableWhitespace` to false. This flag allows the user to specify whether a validating parser should include ignorable whitespaces as text nodes. It has no effect on non-validating parsers which always include non-markup text.

When set to false, all ignorable whitespace will be discarded and no text node is added to the DOM tree.

TPXPARSCACHESET

Sets `setcacheGrammarFromParse` to true. This flag allows users to enable or disable caching of grammar when parsing XML documents. When set to true, the parser will cache the resulting grammar for use in subsequent parses. If the flag is set to true, the *Use cached grammar* flag will also be set to true.

TPXPARSCACHERESET

Resets `resetCachedGrammarPool`. Resets the documents vector pool and release all the associated memory back to the system.

When parsing a document using a DOM parser, all memory allocated for a DOM tree is associated to the DOM document.

If you do multiple parse using the same DOM parser instance, then multiple DOM documents will be generated and saved in a vector pool. All these documents (and thus all the allocated memory) won't be deleted until the parser instance is destroyed.

If you do not need these DOM documents anymore and do not want to destroy the DOM parser instance at this moment, then you can call this method to reset the document vector pool and release all the allocated memory back to the system. It is an error to call this method if you are in the middle of a parse (for example, a mid progressive parse).

## Return Values

Upon success, `tpxmltofml ()` returns a 0. This function returns -1 on error and sets `tperrno` to indicate the error condition.

## Errors

The function may fail for the following reasons.

[TPEINVAL]
: Either fml32bufp or xmlbufp is not a valid typed buffer, or parser has problems understanding the input.

[TPESYSTEM]
: A Tuxedo system error has occurred. The exact nature of the error is written to userlog(3c). This will also indicate when a conversion to FML was unable to be done. In that instance error detail info will be added to the userlog.

[TPEOS]
: An operating system error has occurred. A numeric value representing the system call that failed is available in Uunixerr.

### SEE ALSO

- tpxmltofml32(3c), tpfml32toxml(3c), tpfmltoxml(3c)
- Converting XML Data To and From FML/FML32 Buffers

## TRY(3c)

### Name

TRY()—Exception-returning interface.

### Synopsis

```
#include <texc.h>

TRY
try_block
[  CATCH(exception_name) handler_block] ...
[CATCH_ALL handler_block]
ENDTRY

TRY
try_block
FINALLY
finally_block
ENDTRY

RAISE(exception_name)
```

```
RERAISE

/* declare exception */
EXCEPTION exception_name;

/* initialize address (application) exception */
EXCEPTION_INIT(EXCEPTION exception_name)

/* intialize status exception (map status to exception */
exc_set_status(EXCEPTION *exception_name, long status)

/* map status exception to status */
exc_get_status(EXCEPTION *exception_name, long *status)

/* compare exceptions */
exc_matches(EXCEPTION *e1, EXCEPTION *e2)

/* print error to stderr */
void exc_report(EXCEPTION *exception)
```

## Description

The TRY/CATCH interface provides a mechanism to handle exceptions without the use of status variables (for example, *errno* or status variables passed back from an RPC operation). These macros are defined in texc.h.

The TRY *try_block* is a block of C or C++ declarations and statements in which an exception may be raised (code that is not associated with raising an exception should be placed before or after the *try_block*). Each TRY/ENDTRY pair constitutes a "scope," with respect to exceptions (not unlike C scoping), or a region of code over which exceptions are caught. These scopes can be properly nested. When an exception is raised, an error is reported to the application by searching the active scopes for actions written to handle ("absorb") an exception (CATCH or CATCH_ALL clauses) or complete the scopes (FINALLY clauses). If a scope does not handle an exception, the scope is torn down with the exception raised at the next higher level (unwinding the stack of exception scopes). Execution resumes at the point after which the exception is handled; there is no provision for resuming execution at the point of error. If the exception is not handled by any scope, the program is terminated (a message is written to the log via userlog(3c) and abort(3) is called).

Zero or more occurrences of CATCH (*exception_name*) *handler_block* may be provided. Each *handler_block* is a block of C or C++ declarations and statements in which the associated exception (*exception_name*) is processed (normally, actions are specified for recovery from the failure). If an exception is raised by a statement in *try_block*, then the first CATCH clause that matches the exception is executed.

Within a CATCH or CATCH_ALL *handler_block*, the current exception can be referenced by the EXCEPTION pointer THIS_CATCH (for example, for logic based on or printing the exception value).

If the exception is not handled by one of the CATCH clauses, then the CATCH_ALL clause is executed. By default, no further action is taken for an exception that is handled by a CATCH or CATCH_ALL clause. If no CATCH_ALL clause exists, then the exception is raised at the *try_block* at the next higher level, assuming that the *try_block* is nested within another *try_block*. If an ANSI C compiler is used, register and automatic variables that are used in the handler blocks should be declared with the volatile attribute (as is true of any blocks that use setjmp/longjmp). Also note that output parameters and return values from the functions that can generate an exception are indeterminate.

Within a CATCH or CATCH_ALL *handler_block*, the current exception can be propagated to the next higher level (the exception is "reraised") using the RERAISE statement. The RERAISE statement must appear lexically within the scope of a *handler_block* (that is, not within a function called by the *handler_block*). Any exception that is caught but not fully handled should be reraised. In many cases, a CATCH_ALL handler should reraise the exception because the handler is not written to handle every exception. The application should also be written such that an exception is raised to the proper scope such that the handler blocks take the appropriate actions and modify the appropriate state (for example, if an exception occurs while opening a file, the handler function for that level should not try to close the unopened file).

An exception can be raised from anywhere by using the RAISE(*exception_name*) statement. This statement causes the exception to start propagating at the current *try_block* and will be reraised until it is handled.

The FINALLY clause can be used to specify an epilogue block of code that is executed after the *try_block*, whether or not an exception is raised. If an exception is raised in the *try_block*, it is reraised after the *finally_block* is executed. This clause can be used to avoid replicating epilogue code twice, once in a CATCH_ALL clause, and again after the ENDTRY. It is normally used to execute cleanup activities, restoring invariants (for example, shared data, locks) as the scopes are unwound, whether or not exceptions are raised (that is, on both normal and abnormal exits from the block). Note (in the "Synopsis" section) that a FINALLY clause cannot be used with a CATCH or CATCH_ALL clause for the same *try_block*; use nested *try_block*s.

The ENDTRY statement must be used to complete the TRY block, since it contains code that must be executed to make sure that exceptions are handled and the context is cleaned up. A *try_block*, *handler_block*, or *finally_block* must not contain a return, non-local jump, or any other means of leaving the block such that the ENDTRY is not reached (for example, goto(), break(), continue(), longjmp()).

This interface is provided to handle exceptions from RPC operations. However, this is a generic interface that can be used for any application. An exception is declared to be of type EXCEPTION. (This is a complex data type; do not try to use it like a long integer.) There are two types of exceptions. They are declared in the same manner but initialized differently.

One type of exception is used to define application exceptions. It is initialized by calling the EXCEPTION_INIT() macro. The address of the exception is stored as the value within the *address* exception. Note that this value is valid only within a single address space and will change if the exception is an automatic variable. For this reason, an *address* exception should be declared as a static or external variable, not an automatic or register variable. The exc_get_status() macro will evaluate to -1 for an *address* exception. Using the exc_set_status() macro on this exception will make it a *status* exception.

The exc_matches macro can be used to compare two exceptions. To compare equal, the exceptions must both be the same type and have the same value (for example, the same status value for *status* exceptions, or the same addresses for *address* exceptions). This comparison is used for the CATCH clause, described above.

When status exceptions are raised, a common part of handling the exception might be to print out the status value, or better yet, a string indicating what status value occurred. If the string is to be printed to the standard error output, then the function exc_report() can be called with a pointer to the *status* exception to print the string in one operation.

```
CATCH_ALL
{
 exc_report(THIS_CATCH);
}
ENDTRY
```

If something else is to be done with the string (for example, printing the error to the user log), exc_get_status() can be used on THIS_CATCH to get the status value (remember that THIS_CATCH is already a pointer to an EXCEPTION, not an EXCEPTION), and dce_error_inq_text() can be used to get the string value associated with the status value.

```
CATCH_ALL
{
```

```
  unsigned long status_to_convert;
  unsigned char error_text[200];
  int status;

  exc_get_status(THIS_CATCH,status_to_convert);
  dce_error_inq_text(status_to_convert, error_text, status);
  userlog("%s", (char *)error_text);
}
ENDTRY
```

**Note:** A thread in a multithreaded application may invoke the TRY/CATCH interface while running in any context state, including TPINVALIDCONTEXT.

## When to Use Exception and Status Returns

The status of RPC operations can be determined portably by defining status variables for each operation ([comm_status] and [fault_status] parameters are defined via the Attribute Configuration File). The status-returning interface is the only interface provided in the X/OPEN RPC specification. The fault_status attribute indicates that errors occurring on the server due to incorrectly specified parameter values, resource constraints, or coding errors be reported by an additional status argument or return value. Similarly, the comm_status attribute indicates that RPC communications failures be reported by an additional status argument or return value. Using status values works well for fine-grained error handling (on a per-call basis) with recovery specified for each possible error on each call, and where it is necessary to retry from the point of failure. The disadvantage is that it is not transparent whether or not the call is local or remote. The remote call has additional status parameters, or a status return value instead of being a void return. Thus, the application must have procedure declarations adjusted between local and distributed code.

For application portability from an OSF/DCE environment, the TRY/CATCH exception-returning interface is also provided. This interface may not be provided in all environments. However, it has the advantage that procedure declarations need not be adjusted between local and distributed code, maintaining existing interfaces. The checking for errors can be simplified such that each procedure call does not have specific failure checking or recovery code. If an error is not handled at some level, then the program exits with a system error message such that the error is detected and can be corrected (omissions become more obvious). Exceptions work better for coarse-grained exception handling.

## Built-in Exceptions

The exceptions shown in Table 12 are “built-in” to the use of this exception interface. The first TRY clause sets up a signal handler to catch the signals list below if they are not currently ignored or caught; the other exceptions are defined only for DCE program portability.

**Table 12 Built-in Exceptions**

| Exception | Description |
|---|---|
| exc_e_SIGBUS | An unhandled SIGBUS signal occurred. |
| exc_e_SIGEMT | An unhandled SIGEMT signal occurred. |
| exc_e_SIGFPE | An unhandled SIGFPE signal occurred. |
| exc_e_SIGILL | An unhandled SIGILL signal occurred. |
| exc_e_SIGIOT | An unhandled SIGIOT signal occurred. |
| exc_e_SIGPIPE | An unhandled SIGPIPE signal occurred. |
| exc_e_SIGSEGV | An unhandled SIGSEGV signal occurred. |
| exc_e_SIGSYS | An unhandled SIGSYS signal occurred. |
| exc_e_SIGTRAP | An unhandled SIGTRAP signal occurred. |
| exc_e_SIGXCPU | An unhandled SIGXCPU signal occurred. |
| exc_e_SIGXFSZ | An unhandled SIGXFSZ signal occurred. |
| pthread_e_badparam | |
| pthread_e_defer_q_full | |
| pthread_e_existence | |
| pthread_e_in_use | |
| pthread_e_nostackmem | |
| pthread_e_nostack | |
| pthread_e_signal_q_full | |
| pthread_e_stackovf | |

**Table 12 Built-in Exceptions (Continued)**

| Exception | Description |
|---|---|
| pthread_e_unimp | |
| pthread_e_use_error | |
| exc_e_decovf | |
| exc_e_exquota | |
| exc_e_fltdiv | |
| exc_e_fltovf | |
| exc_e_fltund | |
| exc_e_illaddr | |
| exc_e_insfmem | |
| exc_e_intdiv | |
| exc_e_intovf | |
| exc_e_nopriv | |
| exc_e_privinst | |
| exc_e_resaddr | |
| exc_e_resoper | |
| exc_e_subrng | |
| exc_e_uninitexc | |

These same exception codes are also defined with the "_e" at the end of the name (for example, exc_e_SIGBUS is also defined as exc_SIGBUS_e). Equivalent status codes are defined with similar names but the "_e_" is changed to "_s_" (for example, exc_e_SIGBUS is equivalent to the exc_s_SIGBUS status code).

## Caveats

In OSF/DCE, the header file is named exc_handling.h; the BEA Tuxedo ATMI system header file is texc.h. It is not possible for the same source file to use both DCE and BEA Tuxedo ATMI

system exception handling. Further, within a program, the handling of signal exceptions can only be done by either DCE or the BEA Tuxedo ATMI system, not both.

## Examples

The following is an example C source file that uses exceptions:

```
#include <texc.h>

EXCEPTION badopen_e;                      /* declare exception for bad open() */

doit(char *filename)
{
  EXCEPTION_INIT(badopen_e);               /* initialize exception */
  TRY  get_and_update_data(filename);      /* do the operation */
  CATCH(badopen_e)                         /* exception - open() failed */
     fprintf(stderr, "Cannot open %s\en", filename);
  CATCH_ALL                                /* handle other errors */
     /* handle rpc service not available, ... */
     exc_report(THIS_CATCH)
   ENDTRY
}
/*
 * Open output file
 * Get the remote data item
 * Write out to file
 */
get_and_update_data(char *filename)
{
  FILE *fp;
  if ((fp == fopen(filename)) == NULL) /* open output file */
    RAISE(badopen_e);                  /* raise exception */
  TRY
    /* in this block, file is opened successfully -
     * use associated FINALLY to close file
     */
      long data;
      /*
       * Execute RPC call - exceptions are raised to the calling
       * function, doit()
       */
      data = remote_get_data();
      fprintf(fp, "%ld\en", data);
  FINALLY
    /* Whether or not exceptions are raised, close the file */
     fclose(fp);
  ENDTRY
}
```

### See Also

`userlog(3c)`

`abort`(2) in a UNIX system reference manual

*Programming BEA Tuxedo ATMI Applications Using TxRPC*

## tuxgetenv(3c)

### Name

`tuxgetenv()`—Returns value for environment name.

### Synopsis

```
#include <atmi.h>
char *tuxgetenv(char *name)
```

### Description

`tuxgetenv()` searches the environment list for a string of the form *`name=value`* and, if the string is present, returns a pointer to the *`value`* in the current environment. Otherwise, it returns a NULL pointer.

This function provides a portable interface to environment variables across the different platforms on which the BEA Tuxedo ATMI system is supported, including those platforms that do not normally have environment variables.

Note that `tuxgetenv` is case-sensitive.

A thread in a multithreaded application may issue a call to `tuxgetenv()` while running in any context state, including `TPINVALIDCONTEXT`.

### Return Values

If a pointer to the string exists, `tuxgetenv()` returns that pointer. If a pointer does not exist, `tuxgetenv()` returns a NULL pointer.

### Portability

On MS Windows, this function overcomes the inability to share environment variables between an application and a Dynamic Link Library. The BEA Tuxedo ATMI Workstation DLL maintains an environment copy for each application that is attached to it. This associated environment and context information is destroyed when `tpterm()` is called from a Windows

application. The value of an environment variable could be changed after the application program calls `tpterm()`.

It is recommended that uppercase variable names be used for the Windows environments. (`tuxreadenv()` converts all environment variable names to uppercase.)

### See Also

`tuxputenv(3c)`, `tuxreadenv(3c)`

## tuxgetmbaconv(3c)

### Name

`tuxgetmbaconv()`—Gets the value for environment variable `TPMBACONV` in the process environment.

### Synopsis

```
#include <atmi.h>
extern int tperrno;

int
tuxgetmbaconv(long flags)     /* Get TPMBACONV info */
```

### Description

This function is used for getting the TPMBACONV status. The `tuxgetnombaconv()` function is used by an application developer to check if the automatic conversion capability of the typed switch buffers is turned off. By default the TPMBACONV is not set and automatic conversion functions are used.

The *`flags`* argument is not currently used and should be set to 0.

### Return Values

`tuxgetnombaconv()` returns MBAUTOCONVERSION_ON if the TPMBACONV is set and MBAUTOCONVERSION_OFF if TPMBACONV is not set.

### See Also

`tuxgetenv(3c)`, `tuxsetmbaconv(3c)`

## tuxgetmbenc(3c)

### Name

`tuxgetmbenc()`—Gets the code-set encoding name for environment variable `TPMBENC` in the process environment.

### Synopsis

```
#include <atmi.h>
extern int tperrno;

int
tuxgetmbenc(char *enc_name, long flags)
```

### Description

This function is used for getting the codeset encoding name that is contained in the TPMBENC environment variable. This environment variable is automatically used as the default codeset encoding name when an MBSTRING typed buffer is created. The default encoding name can be reset or unset using the tpsetmbenc() function once the new message is available.

The *`enc_name`* argument will contain the value of the TPMBENC environment variable upon successful execution of this function. This pointer should be large enough for the encoding name to be copied into.

The *`flags`* argument is not currently used and should be set to 0.

### Return Values

Upon success, tuxgetmbenc() returns 0; otherwise, it returns a non-zero value on error.

### See Also

`tpconvmb(3c)`, `tpgetmbenc(3c)`, `tpsetmbenc(3c)`, `tuxgetenv(3c)`, `tuxsetmbenc(3c)`

## tuxputenv(3c)

### Name

`tuxputenv()`—Changes or adds a value to the environment.

### Synopsis

```
#include <atmi.h>
int tuxputenv(char *string)
```

### Description

*string* points to a string of the form "name=value." tuxputenv() makes the value of the environment variable name equal to value by altering an existing variable or creating a new one. In either case, the string pointed to by *string* becomes part of the environment.

This function provides a portable interface to environment variables across the different platforms on which the BEA Tuxedo ATMI system is supported, including those platforms that do not normally have environment variables.

Note that tuxputenv() is case-sensitive.

A thread in a multithreaded application may issue a call to tuxputenv() while running in any context state, including TPINVALIDCONTEXT.

### Return Values

If tuxputenv() cannot obtain enough space, via malloc(), for an expanded environment, it returns a non-zero integer. Otherwise, it returns zero.

### Portability

On MS Windows, this function overcomes the inability to share environment variables between an application and a Dynamic Link Library. The BEA Tuxedo ATMI system Workstation DLL maintains an environment copy for each application that is attached to it. This associated environment and context information is destroyed when tpterm() is called from a Windows application. The value of an environment variable could be changed after the application program calls tpterm().

We recommend using uppercase variable names for the DOS, Windows, and OS/2, environments. (tuxreadenv() converts all environment variable names to uppercase.)

### See Also

tuxgetenv(3c), tuxreadenv(3c)

## tuxreadenv(3c)

### Name

tuxreadenv()—Adds variables to the environment from a file.

## Synopsis

```
#include <atmi.h>
int tuxreadenv(char *file, char *label)
```

## Description

`tuxreadenv()` reads a file containing environment variables and adds them to the environment, independent of platform. These variables are available using `tuxgetenv()` and can be reset using `tuxputenv()`.

The format of the environment file is as follows:

- Any leading space or tab character on a line is ignored and is not considered in the following points.
- Lines containing variables to be put into the environment are of the form:

  *variable*=*value*

  or

  set *variable*=*value*

  where *variable* must begin with an alphabetic or underscore character and contain only alphanumeric or underscore characters, and *value* may contain any character except newline.
- Within the *value*, strings of the form ${*env*} are expanded using variables already in the environment (forward referencing is not supported and if a value is not set, the variable is replaced with the empty string). Backslash (\) may be used to escape the dollar sign and itself. All other shell quoting and escape mechanisms are ignored and the expanded *value* is placed into the environment.
- Lines beginning with slash (/), pound sign (#), semicolon (;), or exclamation point (!) are treated as comments and ignored. Lines beginning with other characters besides these comment characters, a left square bracket, or an alphabetic or underscore character are reserved for future use; their use is undefined.

- The file is partitioned into sections by lines beginning with left square bracket ([), which acts as a label. The label will be silently truncated if longer than 31 characters. The format of a label is:

  `[`*`label`*`]`

  where *`label`* follows the same rules for *`variable`* above (lines with invalid *`label`* values are ignored).

- Variable lines between the top of the file and the first label are put into the environment for all labels (this is the global section). Other variables are put into the environment only if the label matches the label specified for the application. A label of [] will indicate the global section.

If *`file`* is NULL, then a default filename is used. The fixed filenames are as follows:

```
DOS, Windows, OS2, NT: C:\TUXEDO\TUXEDO.ENV
MAC: TUXEDO.ENV in the system preferences directory
NETWARE: SYS:SYSTEM\TUXEDO.ENV
POSIX: /usr/tuxedo/TUXEDO.ENV or /var/opt/tuxedo/TUXEDO.ENV
```

If *`label`* is NULL, then only variables in the global section are put into the environment. For other values of *`label`*, the global section variables plus any variables in a section matching the *`label`* are put into the environment.

An error message is printed to the `userlog()` if there is a memory failure, if a non-NULL filename does not exist, or if a non-NULL label does not exist.

A thread in a multithreaded application may issue a call to `tuxreadenv()` while running in any context state, including `TPINVALIDCONTEXT`.

## Example

The following is an example environment file.

```
TUXDIR=/usr/tuxedo
[application1]
;this is a comment
/* this is a comment */
#this is a comment
//this is a comment
FIELDTBLS=app1_flds
FLDTBLDIR=/usr/app1/udataobj
[application2]
FIELDTBLS=app2_flds
FLDTBLDIR=/usr/app2/udataobj
```

### Return Values

If `tuxreadenv()` cannot obtain enough space, via `malloc()`, for an expanded environment, or if it cannot open and read a file with a non-NULL name, it returns a non-zero integer. Otherwise, `tuxreadenv()` returns zero.

### Portability

In the DOS, Windows, OS/2, and NetWare environments, `tuxreadenv()` converts all environment variable names to uppercase.

### See Also

`tuxgetenv(3c)`, `tuxputenv(3c)`

## tuxsetmbaconv(3c)

### Name

`tuxsetmbaconv()`—Sets the value for environment variable `TPMBACONV` in the process environment.

### Synopsis

```
#include <atmi.h>
extern int tperrno;

int
tuxsetmbaconv(int onoff, long flags)     /* Set TPMBACONV */
```

### Description

This function is used for setting or resetting the TPMBACONV environment variable. By default TPMBACONV is not set and automatic conversion functions are used.

The *`onoff`* argument is equal to MBAUTOCONVERSION_OFF to unset TPMBACONV and turn off auto-conversions. It is equal to MBAUTOCONVERSION_ON to set TPMBACONV and turn on the typed switch buffers auto-conversion of codeset multi-byte data.

The *`flags`* argument is not currently used and should be set to 0.

### Return Values

Upon success, `tuxsetnombaconv()` returns 0; otherwise, it returns a non-zero value on error. (e.g. it returns -1 if the *`onoff`* arg is not one of the defined values).

### See Also

`tuxgetmbaconv(3c)`, `tuxputenv(3c)`

## tuxsetmbenc(3c)

### Name

`tuxsetmbenc()`—Sets the code-set encoding name for environment variable `TPMBENC` in the process environment.

### Synopsis

```
#include <atmi.h>
extern int tperrno;

int
tuxsetmbenc(char *enc_name, long flags)
```

### Description

This function is used for setting or resetting the codeset encoding name that is contained in the TPMBENC environment variable. This environment variable is automatically used as the default codeset encoding name when an MBSTRING typed buffer is created. This default encoding name can be reset or unset using the tpsetmbenc() function once the new message is available.

The *`enc_name`* argument is the encoding name to use to identify the codeset.

The *`flags`* argument is not currently used and should be set to 0.

### Return Values

Upon success, tuxsetmbenc() returns 0; otherwise, it returns a non-zero value on error.

### See Also

`tpconvmb(3c)`, `tpgetmbenc(3c)`, `tpsetmbenc(3c)`, `tuxgetmbenc(3c)`, `tuxputenv(3c)`

## tx_begin(3c)

### Name

`tx_begin()`—Begins a global transaction.

## Synopsis

```
#include <tx.h>
int tx_begin(void)
```

## Description

`tx_begin()` is used to place the calling thread of control in transaction mode. The calling thread must first ensure that its linked resource managers have been opened (via `tx_open()`) before it can start transactions. `tx_begin()` fails (returning [`TX_PROTOCOL_ERROR`]) if the caller is already in transaction mode or `tx_open()` has not been called.

Once in transaction mode, the calling thread must call `tx_commit()` or `tx_rollback()` to complete its current transaction. There are certain cases related to transaction chaining where `tx_begin()` does not need to be called explicitly to start a transaction. See `tx_commit()` and `tx_rollback()` for details.

In a multithreaded application, a thread in the `TPINVALIDCONTEXT` state is not allowed to issue a call to `tx_begin()`.

## Optional Set-up

`tx_set_transaction_timeout()`

## Return Value

Upon successful completion, `tx_begin()` returns `TX_OK`, a non-negative return value.

## Errors

Under the following conditions, `tx_begin()` fails and returns one of these negative values:

**[**`TX_OUTSIDE`**]**
: The transaction manager is unable to start a global transaction because the calling thread of control is currently participating in work outside any global transaction with one or more resource managers. All such work must be completed before a global transaction can be started. The caller's state with respect to the local transaction is unchanged.

**[**`TX_PROTOCOL_ERROR`**]**
: The function was called in an improper context (for example, the caller is already in transaction mode). The caller's state with respect to transaction mode is unchanged.

**[**`TX_ERROR`**]**
: Either the transaction manager or one or more of the resource managers encountered a transient error trying to start a new transaction. When this error is returned, the caller is not in transaction mode. The exact nature of the error is written to a log file.

[TX_FAIL]
: Either the transaction manager or one or more of the resource managers encountered a fatal error. The nature of the error is such that the transaction manager and/or one or more of the resource managers can no longer perform work on behalf of the application. When this error is returned, the caller is not in transaction mode. The exact nature of the error is written to a log file.

### See Also

tx_commit(3c), tx_open(3c), tx_rollback(3c), tx_set_transaction_timeout(3c)

### Warnings

XA-compliant resource managers must be successfully opened to be included in the global transaction. (See tx_open(3c) for details.) Both the X/Open TX interface and the X-Windows system define the type XID. It is not possible to use both X-Windows calls and TX calls in the same file.

## tx_close(3c)

### Name

tx_close()—Closes a set of resource managers.

### Synopsis

```
#include <tx.h>
int tx_close(void)
```

### Description

tx_close() closes a set of resource managers in a portable manner. It invokes a transaction manager to read resource-manager-specific information in a transaction-manager-specific manner and pass this information to the resource managers linked to the caller.

tx_close() closes all resource managers to which the caller is linked. This function is used in place of resource-manager-specific “close” calls and allows an application program to be free of calls which may hinder portability. Since resource managers differ in their termination semantics, the specific information needed to “close” a particular resource manager must be published by each resource manager.

tx_close() should be called when an application thread of control no longer wishes to participate in global transactions. tx_close() fails (returning [TX_PROTOCOL_ERROR]) if the

caller is in transaction mode. That is, no resource managers are closed even though some may not be participating in the current transaction.

When `tx_close()` returns success (`TX_OK`), all resource managers linked to the calling thread are closed.

In a multithreaded application, a thread in the `TPINVALIDCONTEXT` state is not allowed to issue a call to `tx_close()`.

## Return Value

Upon successful completion, `tx_close`() returns `TX_OK`, a non-negative return value.

## Errors

Under the following conditions, `tx_close()` fails and returns one of these negative values:

**[**`TX_PROTOCOL_ERROR`**]**
The function was called in an improper context (for example, the caller is in transaction mode). No resource managers are closed.

**[**`TX_ERROR`**]**
Either the transaction manager or one or more of the resource managers encountered a transient error. The exact nature of the error is written to a log file. All resource managers that could be closed are closed.

**[**`TX_FAIL`**]**
Either the transaction manager or one or more of the resource managers encountered a fatal error. The nature of the error is such that the transaction manager and/or one or more of the resource managers can no longer perform work on behalf of the application. The exact nature of the error is written to a log file.

## See Also

`tx_open(3c)`

## Warnings

Both the X/Open TX interface and the X-Windows system define the type XID. It is not possible to use both X-Windows calls and TX calls in the same file.

# tx_commit(3c)

## Name

tx_commit()—Commits a global transaction.

## Synopsis

```
#include <tx.h>
int tx_commit(void)
```

## Description

tx_commit() is used to commit the work of the transaction active in the caller's thread of control.

If the *transaction_control* characteristic (see tx_set_transaction_control(3c)) is TX_UNCHAINED, then when tx_commit() returns, the caller is no longer in transaction mode. However, if the *transaction_control* characteristic is TX_CHAINED, then when tx_commit() returns, the caller remains in transaction mode on behalf of a new transaction (see the Return Value and Errors sections below).

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tx_commit().

## Optional Set-up

- tx_set_commit_return()
- tx_set_transaction_control()
- tx_set_transaction_timeout()

## Return Value

Upon successful completion, tx_commit() returns TX_OK, a non-negative return value.

## Errors

Under the following conditions, tx_commit() fails and returns one of these negative values:

**[**TX_NO_BEGIN**]**

The current transaction committed successfully; however, a new transaction could not be started and the caller is no longer in transaction mode. This return value may occur only when the *transaction_control* characteristic is TX_CHAINED.

**[**TX_ROLLBACK**]**
The current transaction could not commit and has been rolled back. In addition, if the *transaction_control* characteristic is TX_CHAINED, a new transaction is started.

**[**TX_ROLLBACK_NO_BEGIN**]**
The transaction could not commit and has been rolled back. In addition, a new transaction could not be started and the caller is no longer in transaction mode. This return value can occur only when the *transaction_control* characteristic is TX_CHAINED.

**[**TX_MIXED**]**
The work done on behalf of the transaction was partially committed and partially rolled back. In addition, if the *transaction_control* characteristic is TX_CHAINED, a new transaction is started.

**[**TX_MIXED_NO_BEGIN**]**
The work done on behalf of the transaction was partially committed and partially rolled back. In addition, a new transaction could not be started and the caller is no longer in transaction mode. This return value can occur only when the *transaction_control* characteristic is TX_CHAINED.

**[**TX_HAZARD**]**
Due to a failure, some of the work done on behalf of the transaction may have been committed and some of it may have been rolled back. In addition, if the *transaction_control* characteristic is TX_CHAINED, a new transaction is started.

**[**TX_HAZARD_NO_BEGIN**]**
Due to a failure, some of the work done on behalf of the transaction may have been committed and some of it may have been rolled back. In addition, a new transaction could not be started and the caller is no longer in transaction mode. This return value can occur only when the *transaction_control* characteristic is TX_CHAINED.

**[**TX_PROTOCOL_ERROR**]**
The function was called in an improper context (for example, the caller is not in transaction mode). The caller's state with respect to transaction mode is not changed.

**[**TX_FAIL**]**
Either the transaction manager or one or more of the resource managers encountered a fatal error. The nature of the error is such that the transaction manager and/or one or more of the resource managers can no longer perform work on behalf of the application. The exact nature of the error is written to a log file. The caller's state with respect to the transaction is unknown.

### See Also

`tx_begin(3c)`, `tx_set_commit_return(3c)`, `tx_set_transaction_control(3c)`, `tx_set_transaction_timeout(3c)`

### Warnings

Both the X/Open TX interface and the X-Windows system define the type XID. It is not possible to use both X-Windows calls and TX calls in the same file.

## tx_info(3c)

### Name

`tx_info()`—Returns global transaction information.

### Synopsis

```
#include <tx.h>
int tx_info(TXINFO *info)
```

### Description

`tx_info()` returns global transaction information in the structure pointed to by *`info`*. In addition, this function returns a value indicating whether the caller is currently in transaction mode or not. If *`info`* is non-NULL, then `tx_info()` populates a `TXINFO` structure pointed to by *`info`* with global transaction information. The `TXINFO` structure contains the following elements:

```
XID                 xid;
COMMIT_RETURN       when_return;
TRANSACTION_CONTROL transaction_control;
TRANSACTION_TIMEOUT transaction_timeout;
TRANSACTION_STATE    transaction_state;
```

If `tx_info()` is called in transaction mode, then *`xid`* will be populated with a current transaction branch identifier and *`transaction_state`* will contain the state of the current transaction. If the caller is not in transaction mode, *`xid`* will be populated with the NULL XID (see the `tx.h` file for details). In addition, regardless of whether the caller is in transaction mode, *`when_return`*, *`transaction_control`*, and *`transaction_timeout`* contain the current settings of the *`commit_return`* and *`transaction_control`* characteristics, and the transaction timeout value in seconds.

The transaction timeout value returned reflects the setting that will be used when the next transaction is started. Thus, it may not reflect the timeout value for the caller's current global

transaction since calls made to `tx_set_transaction_timeout()` after the current transaction was begun may have changed its value.

If *`info`* is NULL, no `TXINFO` structure is returned.

In a multithreaded application, a thread in the `TPINVALIDCONTEXT` state is not allowed to issue a call to `tx_info()`.

### Return Value

If the caller is in transaction mode, then 1 is returned. If the caller is not in transaction mode, then 0 is returned.

### Errors

Under the following conditions, `tx_info()` fails and returns one of these negative values:

**[**`TX_PROTOCOL_ERROR`**]**
The function was called in an improper context (for example, the caller has not yet called `tx_open()`).

**[**`TX_FAIL`**]**
The transaction manager encountered a fatal error. The nature of the error is such that the transaction manager can no longer perform work on behalf of the application. The exact nature of the error is written to a log file.

### See Also

`tx_open(3c)`, `tx_set_commit_return(3c)`, `tx_set_transaction_control(3c)`, `tx_set_transaction_timeout(3c)`

### Warnings

Within the same global transaction, subsequent calls to `tx_info()` are guaranteed to provide an XID with the same *`gtrid`* component, but not necessarily the same *`bqual`* component. Both the X/Open TX interface and the X-Windows system define the type XID. It is not possible to use both X-Windows calls and TX calls in the same file.

## tx_open(3c)

### Name

`tx_open()`—Opens a set of resource managers.

## Synopsis

```
#include <tx.h>
int tx_open(void)
```

## Description

`tx_open()` opens a set of resource managers in a portable manner. It invokes a transaction manager to read resource-manager-specific information in a transaction-manager-specific manner and pass this information to the resource managers linked to the caller.

`tx_open()` attempts to open all resource managers that have been linked with the application. This function is used in place of resource-manager-specific "open" calls and allows an application program to be free of calls which may hinder portability. Since resource managers differ in their initialization semantics, the specific information needed to "open" a particular resource manager must be published by each resource manager.

If `tx_open()` returns `TX_ERROR`, then no resource managers are open. If `tx_open()` returns `TX_OK`, some or all of the resource managers have been opened. Resource managers that are not open will return resource-manager-specific errors when accessed by the application. `tx_open()` must successfully return before a thread of control participates in global transactions.

Once `tx_open()` returns success, subsequent calls to `tx_open()` (before an intervening call to `tx_close()`) are allowed. However, such subsequent calls will return success, and the TM will not attempt to reopen any RMs.

In a multithreaded application, a thread in the `TPINVALIDCONTEXT` state is not allowed to issue a call to `tx_open()`.

## Return Value

Upon successful completion, `tx_open()` returns `TX_OK`, a non-negative return value.

## Errors

Under the following conditions, `tx_open()` fails and returns one of these negative values:

**[**`TX_ERROR`**]**
: Either the transaction manager or one or more of the resource managers encountered a transient error. No resource managers are open. The exact nature of the error is written to a log file.

**[**`TX_FAIL`**]**
: Either the transaction manager or one or more of the resource managers encountered a fatal error. `TX_FAIL` is returned if `tpinit()` is not called before the call to `tx_open` in a

secure application (SECURITY APP_PW). The nature of the error is such that the transaction manager and/or one or more of the resource managers can no longer perform work on behalf of the application. The exact nature of the error is written to a log file.

### See Also

`tx_close(3c)`

### Warnings

Both the X/Open TX interface and the X-Windows system define the type XID. It is not possible to use both X-Windows calls and TX calls in the same file.

## tx_rollback(3c)

### Name

`tx_rollback()`—Rolls back a global transaction.

### Synopsis

```
#include <tx.h>
int tx_rollback(void)
```

### Description

`tx_rollback()` is used to roll back the work of the transaction active in the caller's thread of control.

If the *`transaction_control`* characteristic (see `tx_set_transaction_control(3c)`) is `TX_UNCHAINED`, then when `tx_rollback()` returns, the caller is no longer in transaction mode. However, if the *`transaction_control`* characteristic is `TX_CHAINED`, then when `tx_rollback()` returns, the caller remains in transaction mode on behalf of a new transaction (see the Return Value and Errors sections below).

In a multithreaded application, a thread in the `TPINVALIDCONTEXT` state is not allowed to issue a call to `tx_rollback()`.

### Optional Set-up

- `tx_set_transaction_control()`
- `tx_set_transaction_timeout()`

## Return Value

Upon successful completion, `tx_rollback()` returns `TX_OK`, a non-negative return value.

## Errors

Under the following conditions, `tx_rollback()` fails and returns one of these negative values:

**[**`TX_NO_BEGIN`**]**
The current transaction rolled back; however, a new transaction could not be started and the caller is no longer in transaction mode. This return value may occur only when the *`transaction_control`* characteristic is `TX_CHAINED`.

**[**`TX_MIXED`**]**
The work done on behalf of the transaction was partially committed and partially rolled back. In addition, if the *`transaction_control`* characteristic is `TX_CHAINED`, a new transaction is started.

**[**`TX_MIXED_NO_BEGIN`**]**
The work done on behalf of the transaction was partially committed and partially rolled back. In addition, a new transaction could not be started and the caller is no longer in transaction mode. This return value can occur only when the *`transaction_control`* characteristic is `TX_CHAINED`.

**[**`TX_HAZARD`**]**
Due to a failure, some of the work done on behalf of the transaction may have been committed and some of it may have been rolled back. In addition, if the *`transaction_control`* characteristic is `TX_CHAINED`, a new transaction is started.

**[**`TX_HAZARD_NO_BEGIN`**]**
Due to a failure, some of the work done on behalf of the transaction may have been committed and some of it may have been rolled back. In addition, a new transaction could not be started and the caller is no longer in transaction mode. This return value can occur only when the *`transaction_control`* characteristic is `TX_CHAINED`.

**[**`TX_COMMITTED`**]**
The work done on behalf of the transaction was heuristically committed. In addition, if the *`transaction_control`* characteristic is `TX_CHAINED`, a new transaction is started.

**[**`TX_COMMITTED_NO_BEGIN`**]**
The work done on behalf of the transaction was heuristically committed. In addition, a new transaction could not be started and the caller is no longer in transaction mode. This return value can occur only when the *`transaction_control`* characteristic is `TX_CHAINED`.

**[**TX_PROTOCOL_ERROR**]**
The function was called in an improper context (for example, the caller is not in transaction mode).

**[**TX_FAIL**]**
Either the transaction manager or one or more of the resource managers encountered a fatal error. The nature of the error is such that the transaction manager and/or one or more of the resource managers can no longer perform work on behalf of the application. The exact nature of the error is written to a log file. The caller's state with respect to the transaction is unknown.

### See Also

tx_begin(3c), tx_set_transaction_control(3c), tx_set_transaction_timeout(3c)

### Warnings

Both the X/Open TX interface and the X-Windows system define the type XID. It is not possible to use both X-Windows calls and TX calls in the same file.

## tx_set_commit_return(3c)

### Name

tx_set_commit_return()—Sets the *commit_return* characteristic.

### Synopsis

```
#include <tx.h>
int tx_set_commit_return(COMMIT_RETURN when_return)
```

### Description

tx_set_commit_return() sets the *commit_return* characteristic to the value specified in *when_return*. This characteristic affects the way tx_commit() behaves with respect to returning control to its caller. tx_set_commit_return() may be called regardless of whether its caller is in transaction mode. This setting remains in effect until changed by a subsequent call to tx_set_commit_return().

The initial setting for this characteristic is TX_COMMIT_COMPLETED.

The following are the valid settings for *when_return*:

TX_COMMIT_DECISION_LOGGED
: This flag indicates that tx_commit() should return after the commit decision has been logged by the first phase of the two-phase commit protocol but before the second phase has completed. This setting allows for faster response to the caller of tx_commit(). However, there is a risk that a transaction will have a heuristic outcome, in which case the caller will not find out about this situation via return codes from tx_commit(). Under normal conditions, participants that promise to commit during the first phase will do so during the second phase. In certain unusual circumstances however (for example, long-lasting network or node failures), phase 2 completion may not be possible and heuristic results may occur.

TX_COMMIT_COMPLETED
: This flag indicates that tx_commit() should return after the two-phase commit protocol has finished completely. This setting allows the caller of tx_commit() to see return codes that indicate that a transaction had or may have had heuristic results.

In a multithreaded application, a thread in the TPINVALIDCONTEXT state is not allowed to issue a call to tx_set_commit_return().

## Return Value

Upon successful completion, tx_set_commit_return() returns TX_OK, a non-negative return value.

## Errors

Under the following conditions, tx_set_commit_return() does not change the setting of the *commit_return* characteristic and returns one of these negative values:

**[**TX_EINVAL**]**
: *when_return* is not one of TX_COMMIT_DECISION_LOGGED or TX_COMMIT_COMPLETED.

**[**TX_PROTOCOL_ERROR**]**
: The function was called in an improper context (for example, the caller has not yet called tx_open()).

**[**TX_FAIL**]**
: The transaction manager encountered a fatal error. The nature of the error is such that the transaction manager can no longer perform work on behalf of the application. The exact nature of the error is written to a log file.

## See Also

tx_commit(3c), tx_info(3c), tx_open(3c)

### Warnings

Both the X/Open TX interface and the X-Windows system define the type XID. It is not possible to use both X-Windows calls and TX calls in the same file.

## tx_set_transaction_control(3c)

### Name

`tx_set_transaction_control()`—Sets the *`transaction_control`* characteristic.

### Synopsis

```
#include <tx.h>
int tx_set_transaction_control(TRANSACTION_CONTROL control)
```

### Description

`tx_set_transaction_control()` sets the *`transaction_control`* characteristic to the value specified in *`control`*. This characteristic determines whether `tx_commit()` and `tx_rollback()` start a new transaction before returning to their caller. `tx_set_transaction_control()` may be called regardless of whether the application program is in transaction mode. This setting remains in effect until changed by a subsequent call to `tx_set_transaction_control()`.

The initial setting for this characteristic is `TX_UNCHAINED`.

The following are the valid settings for *`control`*:

`TX_UNCHAINED`
: This flag indicates that `tx_commit()` and `tx_rollback()` should not start a new transaction before returning to their caller. The caller must issue `tx_begin()` to start a new transaction.

`TX_CHAINED`
: This flag indicates that `tx_commit()` and `tx_rollback()` should start a new transaction before returning to their caller.

In a multithreaded application, a thread in the `TPINVALIDCONTEXT` state is not allowed to issue a call to `tx_set_transaction_control()`.

### Return Value

Upon successful completion, `tx_set_transaction_control()` returns `TX_OK`, a non-negative return value.

### Errors

Under the following conditions, `tx_set_transaction_control()` does not change the setting of the *`transaction_control`* characteristic and returns one of these negative values:

**[**`TX_EINVAL`**]**
*`control`* is not one of `TX_UNCHAINED` or `TX_CHAINED`.

**[**`TX_PROTOCOL_ERROR`**]**
The function was called in an improper context (for example, the caller has not yet called `tx_open()`).

**[**`TX_FAIL`**]**
The transaction manager encountered a fatal error. The nature of the error is such that the transaction manager can no longer perform work on behalf of the application. The exact nature of the error is written to a log file.

### See Also

`tx_begin(3c)`, `tx_commit(3c)`, `tx_info(3c)`, `tx_open(3c)`, `tx_rollback(3c)`

### Warnings

Both the X/Open TX interface and the X-Windows system define the type XID. It is not possible to use both X-Windows calls and TX calls in the same file.

## tx_set_transaction_timeout(3c)

### Name

`tx_set_transaction_timeout()`—Sets the *`transaction_timeout`* characteristic.

### Synopsis

```
#include <tx.h>
int tx_set_transaction_timeout(TRANSACTION_TIMEOUT timeout)
```

### Description

`tx_set_transaction_timeout()` sets the *`transaction_timeout`* characteristic to the value specified in *`timeout`*. This value specifies the time period in which the transaction must complete before becoming susceptible to transaction timeout; that is, the interval between the AP calling `tx_begin()` and `tx_commit()` or `tx_rollback()`. `tx_set_transaction_timeout()` may be called regardless of whether its caller is in transaction mode or not. If

`tx_set_transaction_timeout()` is called in transaction mode, the new `timeout` value does not take effect until the next transaction.

The initial *`transaction_timeout`* value is 0 (no timeout).

*`timeout`* specifies the number of seconds allowed before the transaction becomes susceptible to transaction timeout. It may be set to any value up to the maximum value for a `long` as defined by the system. A *`timeout`* value of zero disables the timeout feature.

In a multithreaded application, a thread in the `TPINVALIDCONTEXT` state is not allowed to issue a call to `tx_set_transaction_timeout()`.

## Return Value

Upon successful completion, `tx_set_transaction_timeout()` returns `TX_OK`, a non-negative return value.

## Errors

Under the following conditions, `tx_set_transaction_timeout()` does not change the setting of the *`transaction_timeout`* characteristic and returns one of these negative values:

**[**`TX_EINVAL`**]**
The timeout value specified is invalid.

**[**`TX_PROTOCOL_ERROR`**]**
The function was called improperly. For example, it was called before the caller called `tx_open()`.

**[**`TX_FAIL`**]**
The transaction manager encountered an error. The nature of the error is such that the transaction manager can no longer perform work on behalf of the application. The exact nature of the error is written to a log file.

## See Also

`tx_begin(3c)`, `tx_commit(3c)`, `tx_info(3c)`, `tx_open(3c)`, `tx_rollback(3c)`

## Warnings

Both the X/Open TX interface and the X-Windows system define the type XID. It is not possible to use both X-Windows calls and TX calls in the same file.

# userlog(3c)

## Name

`userlog()`—Writes a message to the BEA Tuxedo ATMI system central event log.

## Synopsis

```
#include "userlog.h"
extern char *proc_name;

int userlog (format [ ,arg] . . .)
char *format;
```

## Description

`userlog()` accepts a `printf`(3S) style format specification, with a fixed output file—the BEA Tuxedo ATMI system central event log.

The central event log is an ordinary UNIX file whose pathname is composed as follows: If the shell variable `ULOGPFX` is set, its value is used as the prefix for the filename. If `ULOGPFX` is not set, `ULOG` is used. The prefix is determined the first time `userlog()` is called. Each time `userlog()` is called the date is determined, and the month, day, and year are concatenated to the prefix as `mmddyy` to set the name for the file. The first time a process writes to the user log, it first writes an additional message indicating the associated BEA Tuxedo ATMI system version.

The message is then appended to the file. With this scheme, processes that call `userlog()` on successive days will write into different files.

Messages are appended to the log file with a tag made up of the time (`hhmmss`), system name, process name, and process ID, thread ID, and context ID of the calling process. The tag is terminated with a colon (`:`). The name of the process is taken from the pathname of the external variable `proc_name`. If `proc_name` has value NULL, the printed name is set to `?proc`.

BEA Tuxedo ATMI system-generated error messages in the log file are prefixed by a unique identification string of the form:

`<catalog>:number>:`

This string gives the name of the internationalized catalog containing the message string, plus the message number. By convention, BEA Tuxedo ATMI system-generated error messages are used only once, so the string uniquely identifies a location in the source code.

If the last character of the *`format`* specification is not a newline character, `userlog()` appends one.

If the first character of the shell variable ULOGDEBUG is 1 or y, the message sent to userlog() is also written to the standard error of the calling process, using the fprintf(3S) function.

userlog() is used by the BEA Tuxedo ATMI system to record a variety of events.

The userlog mechanism is entirely independent of any database transaction logging mechanism.

A thread in a multithreaded application may issue a call to userlog() while running in any context state, including TPINVALIDCONTEXT.

## Environment Variables

ULOGMILLISEC

An on/off switch environment variable that time stamps messages sent to the userlog file in millisecond time intervals instead of seconds. If not specified, default time stamping is in seconds. The server must be rebooted when ULOGMILLISEC is turned on or off.
Example: ULOGMILLISEC=Y

ULOGRTNSIZE

An on/off switch environment variable that specifies the userlog rotation file size. The default rotation file size is 2GB. The server must be rebooted when ULOGRTNSIZE is turned on or off.
Example: ULOGRTNSIZE=1000000 (when the file size is 1Mb)

Rotated files are saved in using the following syntax: filename.nn.
Example: ULOG.083103.1, ULOG.083103.2 ... ULOG.083103.10, etc.

**Note:** If ULOGRTNSIZE is not specified, file rotation does not take place.

## Portability

The userlog() interface is supported on UNIX and MS-DOS operating systems. The system name produced as part of the log message is not available on MS-DOS systems; therefore, the value PC is used as the system name for MS-DOS systems.

## Examples

If the variable ULOGPFX is set to /application/logs/log and if the first call to userlog() occurred on 9/7/90, the log file created is named /application/logs/log.090790. If the call:

```
userlog("UNKNOWN USER '%s' (UID=%d)", usrname, UID);
```

is made at 4:22:14pm on the UNIX system file named m1 by the sec program, whose process-id is 23431, and the variable usrname contains the string "sxx", and the variable UID contains the integer 123, the following line appears in the log file:

```
162214.m1!sec.23431: UNKNOWN USER 'sxx' (UID=123)
```

If the message is sent to the central event log while the process is in transaction mode, the user log entry has additional components in the tag. These components consist of the literal `gtrid` followed by three `long` hexadecimal integers. The integers uniquely identify the global transaction and make up what is referred to as the global transaction identifier. This identifier is used mainly for administrative purposes, but it does make an appearance in the tag that prefixes the messages in the central event log. If the foregoing message is written to the central event log in transaction mode, the resulting log entry will look like this:

```
162214.logsys!security.23431: gtrid x2 x24e1b803 x239: UNKNOWN USER 'sxx'
(UID=123)
```

If the shell variable `ULOGDEBUG` has a value of `y`, the log message is also written to the standard error of the program named `security`.

### Errors

`userlog()` hangs if the message sent to it is larger than `BUFSIZ` as defined in `stdio.h`

### Diagnostics

`userlog()` returns the number of characters output, or a negative value if an output error was encountered. Output errors include the inability to open, or write to the current log file. Inability to write to the standard error, when `ULOGDEBUG` is set, is not considered an error.

### Notices

It is recommended that applications' use of `userlog()` messages be limited to messages that can be used to help debug application errors; flooding the log with incidental information can make it hard to spot actual errors.

### See Also

- `printf`(3S) in a UNIX system reference manual
- *Using Log Files to Monitor Activity* in *Monitoring Your BEA Tuxedo Application*

## Usignal(3c)

### Name

`Usignal()`—Signal handling in a BEA Tuxedo ATMI system environment.

## Synopsis

```
#include "Usignal.h"

UDEFERSIGS()
UENSURESIGS()
UGDEFERLEVEL()
URESUMESIGS()
USDEFERLEVEL(level)

int (*Usignal(sig,func)()
int sig;
int (*func)();

void Usiginit()
```

## Description

Many of the facilities provided by the BEA Tuxedo ATMI system software require concurrent access to data structures in shared memory. Processes accessing the shared data structures run in user mode, and are thus interruptible by signals sent to them. In order to ensure the consistency of the shared data structures, it is important that the operations which access them not be interrupted by the receipt of certain UNIX signals. The functions described in this section provide protection against the most common signals, and are used internally by much of the BEA Tuxedo ATMI system code. Additionally, they are available to applications to prevent the untimely arrival of a signal.

The idea behind the BEA Tuxedo ATMI system signal handling package is that signals should be deferrable while in critical code sections. To this end, signals are not immediately processed when received. Instead, a BEA Tuxedo ATMI system routine first catches the sent signal. If it is safe to process the signal, the specified action for the signal is taken. If it is not safe to process the signal when it arrives, the arrival is noted, but the processing is deferred until the user indicates that the critical section of code has been terminated.

We recommend against any use of signals in multithreaded programs, although the software does not prevent such usage. If signals are used, however, a thread in a multithreaded application may issue a call to `Usignal()` while running in any context state, including `TPINVALIDCONTEXT`.

## Catching Signals

User code that uses calls `rmopen()` or `tpinit()` should catch signals through the use of the `Usignal()` function. `Usignal()` behaves like the UNIX `signal()` system call, except that

Usignal() first arranges for the signal to be caught by an internal routine before dispatching the user routine.

## Deferring and Restoring Signals

The calls described in this section need only be used if application code wishes to defer signals. In general, these routines are called automatically by BEA Tuxedo ATMI system routines to protect themselves from untimely signal arrival.

Before deferring or restoring signals, the mechanism must be initialized. This is done automatically for BEA Tuxedo ATMI system clients when they call tpinit() and for BEA Tuxedo ATMI system servers. It is also done the first time that the application calls Usignal(). It can be done explicitly by calling Usiginit().

The UDEFERSIGS() macro should be used when entering a section of critical code. After UDEFERSIGS() is called, signals are held in a pending state. The URESUMESIGS() macro should be invoked when the critical section is exited. Note that signal deferrals stack. The stack is implemented via a counter which is initially set to zero. When signals are deferred by a call to UDEFERSIGS(), the counter is incremented. When signals are resumed, by a call to URESUMESIGS(), the counter is decremented. If a signal arrives while the counter is non-zero, the processing of the signal is deferred. If the counter is zero when the signal arrives, the signal is processed immediately. If signal resumption causes the counter to be become zero (that is, prior to the resumption it had value 1), any signals that arrived during the deferral period are processed. In general, each call to UDEFERSIGS() should have a counterpart call to URESUMESIGS().

UDEFERSIGS increments the deferral counter, but returns the value of the counter prior to its incrementation. The macro UENSURESIGS() may be used to explicitly set the deferral counter to zero (and thus force the processing of deferred signals), in case the user wishes to protect against unmatching UDEFERSIGS() and URESUMESIGS().

The function UGDEFERLEVEL() returns the current setting of the deferral counter. The macro USDEFERLEVEL(level) allows the setting of a specific deferral level. UGDEFERLEVEL() and USDEFERLEVEL() are useful to set the counter appropriately in setjmp/longjmp situations where a set of deferrals/resumes are bypassed. The idea is to save the value of the counter when setjmp() is called, via a call to UGDEFERLEVEl(), and to restore it via a call to USDEFERLEVel() when the longjmp() is performed.

## Notices

Usignal provides signal deferral for the following signals: SIGHUP,SIGINT, SIGQUIT, SIGALRM, SIGTERM, SIGUSR1, and SIGUSR2. Handling requests for all other signal numbers are passed directly to signaL() by Usignal(). Signals may be deferred for a considerable time. For this

reason, during signal deferral, individual signal arrivals are counted. When it is safe to process a signal that may have arrived many times, the signal's processing routine is iteratively called to process each arrival of the signal. Before each call the default action for the signal is instantiated. The idea is to handle the deferred occurrences of the signal as if they happened in quick succession in safe code.

In general, users should not mix calls to `signaL()` and `UsignaL()` for the same signal. The recommended procedure is to go through `Usignal()`, so that it is always aware of the state of the signal. Sometimes it may be necessary, such as when an application wants to use alarms within BEA Tuxedo ATMI system services. To do this, `UsiginiT()` should be called to initialize the signal deferring mechanism. Then `signaL()` can be called to override the mechanism for the desired signal. To restore the deferring mechanism for the signal, it is necessary to call `UsignaL()` for the signal with `SIG_IGN`, and then again with the desired signal-handling function.

The shell variable `UIMMEDSIGS` can be used to override the deferral of signals. If the value of this variable begins with the letter `y` as in:

```
UIMMEDSIGS=y
```

signals are not intercepted (and thus not deferred) by the `Usignal()` code. In such a case, a call to `Usignal()` is passed immediately to `signaL()`.

`Usignal` is not available under DOS operating systems.

### Files

`Usignal.h`

### See Also

`signal`(2) in a UNIX system reference manual

## Uunix_err(3c)

### Name

`Uunix_err()`—Prints a UNIX system call error.

### Synopsis

```
#include Uunix.h
```

```
void Uunix_err(s)
char *s;
```

## Description

When a BEA Tuxedo ATMI system function calls a UNIX system call that detects an error, an error is returned. The external integer `Uunixerr()` is set to a value (as defined in `Uunix.h`) that identifies the system call that returned the error. In addition, the system call sets `errno()` to a value (as defined in `errno.h`) that tells why the system call failed.

The `Uunix_err()` function is provided to produce a message on the standard error output, describing the last system call error encountered during a call to a BEA Tuxedo ATMI system function. It takes one argument, a string. The function prints the argument string, then a colon and a blank, followed by the name of the system call that failed, the reason for failure, and a newline. To be of most use, the argument string should include the name of the program that incurred the error. The system call error number is taken from the external variable `Uunixerr()`, the reason is taken from `errno()`. Both variables are set when errors occur. They are not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the array of message strings:

```
extern char *Uunixmsg[];
```

is provided; `Uunixerr()` can be used as an index into this table to get the name of the system call that failed (without the newline).

A thread in a multithreaded application may issue a call to `Uunix_err()` while running in any context state, including `TPINVALIDCONTEXT`.

## Examples

```
#include Uunix.h
extern int Uunixerr, errno;

  ..........
   if((fd=open("myfile", 3, 0660)) == -1)


       {
       Uunixerr = UOPEN;
       Uunix_err("myprog");
       exit(1);
       }
```