BEA TUXEDO

Reference Manual

Section 3C —
C Functions

BEA TUXEDO 6.5 for WLE 5.0
Document Edition 6.5
December 1999

BEA TUXEDO Reference Manual

Document Edition	Date	Software Version
6.5	December 1999	BEA TUXEDO 6.5 for WLE 5.0

About This Document

The TUXEDO 6.5 Reference Manual for WebLogic Enterprise 5.0 includes the following components:

- “Section 1 — Commands” provides information about shell-level commands included with TUXEDO and WebLogic Enterprise (WLE) software.
- “Section 3C — C Functions” describes C language functions that comprise the .Application-Transaction Monitor Interface (ATMI). ATMI provides routines to open and close resources, manage transactions, manage typed buffers, and invoke request/response and conversational service calls.
- “Section 3CBL — COBOL Functions” describes the COBOL bindings for the ATMI interface.
- “Section 3FML — FML Commands” describes C language functions for defining and manipulating Field Manipulation Language (FML) storage structures.
- “Section 5 — File Formats and Data Descriptions” describes various files and tables. This includes the configuration files, `UBBCONFIG` and `TUXCONFIG`, and the TUXEDO Management Information Base (TMIB) classes that provide an interface for managing WLE or TUXEDO systems.

Who Should Use This Document

This document is intended for system administrators and programmers who are interested in creating, configuring, or managing TUXEDO or WebLogic Enterprise applications.

e-docs Web Site

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

For more information about CORBA, Java 2 Enterprise Edition (J2EE), BEA TUXEDO, distributed object computing, transaction processing, C++ programming, and Java programming, see the WLE Bibliography in the WebLogic Enterprise online documentation.

Contact Us!

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

In your e-mail message, please indicate that you are using the documentation for the BEA WebLogic Enterprise 5.0 release.

If you have any questions about this version of BEA WebLogic Enterprise, or if you have problems installing and running BEA WebLogic Enterprise, contact BEA Customer Support through BEA WebSupport at www.beasys.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.

Convention	Item
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.
	Examples:
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Example*s: LPT1 SIGNON OR
`{ }`	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.
	Example:
	`buildobjclient [-v] [-o name ] [-f` *`file-list`*`]...`
	`[-l` *`file-list`*`]...`

Convention	Item
|	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.

`...`	Indicates one of the following in a command line:
	■ That an argument can be repeated several times in a command line
	■ That the statement omits additional optional arguments
	■ That you can enter additional parameters, values, or other information
	The ellipsis itself should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f file-list]...`
	`[-l file-list]...`

. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

Contents

Section 3C — C Functions

intro(3c)

Name `intro`(3c)-introduction to the application-transaction monitor interface.

Description The application-transaction monitor interface provides the interface between the application and the transaction processing system. This interface is known as the ATMI interface. It provides routines to open and close resources, manage transactions, manage typed buffers, and invoke request/response and conversational service calls.

Communication Paradigms The routines 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.

BEA TUXEDO system request /response client/server model 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 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. 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 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 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 either when 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(3c)`.

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 runtime using a routine, `tpsprio(3c)`. By doing so, the caller can override the configuration or default priority when the message is sent.

Conversational Client/server Model

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. While the connection is established, the server is “reserved” such that no other process can establish a connection with the server. 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 can not 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 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 `LIBTUX 1572 “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 system queued message model allows for enqueueing 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 verbs that queue messages and dequeue responses are `tpenqueue(3c)` and `tpdequeue(3c)`. They can be called from any type of BEA TUXEDO system application processes: client, server, or conversational.

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

ATMI Transactions

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

A transaction in the BEA TUXEDO system 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, at possibly 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 `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 (that is, 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 either calling `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 verbs. An application writer may use either set of verbs when writing clients and service routines. In fact, the BEA TUXEDO system does not require all client and server processes within a single application to use one set of verbs or the other. However, the two verb sets may not be used together 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 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 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.

`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 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 `fprint`.

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`.

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. 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 `F_error` or `F_error32`, which is set when errors occur.

`Fstrerror`, and its counterpart, `Fstrerror32`, can be used to retrieve the text of an 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 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 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 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 ‘‘stale.” 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 stale (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 re-issue 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 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.

Beginning in Release 6.4, some additional detail is provided beyond the `TPESVCERR` error code. If a service fails due to exceeding the timeout threshold, an event, `.SysServiecTimeout`, 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 at run time the set of services that it offers, 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 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 it may shrink when it is received into. It depends on the amount of data sent by the sender, and the internal data flow needed to get it from sender to received. Many factors could affect the buffer size, including compression, receiving a message from a different machine type, and the action of the buffer type's `postrecv` function (see `buffer`(3c)). The buffer sizes in /WS 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 expect 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 system guarantees only that a received message will contain all of the data that was sent, not that it will also contain all of the free space.

The process receiving the reply is responsible for noting size changes in the buffer (using `tptypes`) and reallocating it if necessary. All of the BEA TUXEDO system routines that 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 caller's responsibility to free its buffers with `tpfree`. However, in limited cases, the BEA TUXEDO system frees the caller's buffer. Further details about buffer usage are explained in the descriptions of the communication routines.

Buffer Type Switch

The `tmtype_sw_t` structure provides a description necessary when 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 system or by applications adding their own buffer types. These 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 at most 8 characters in length. If this element is not unique in the switch, then `subtype` must be non-NULL.

The element, `subtype`, can be NULL, a string of at most 16 characters, or the wild card 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 wild card 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 that are to be singled out with specific subtype values should appear in the switch before the subtype designated with the wild card. Thus, searching for types and subtypes in the switch is done from top to bottom, and the wild card subtype entry accepts any ``leftover'' type matches.

The element `dfltsize` is used when allocating or re-allocating a buffer. The semantics of `tpalloc` and `tprealloc` are such that the larger of `dfltsize` and the routines' *`size`* parameter is used to create or re-allocate 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 their *`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.

There are four basic buffer types that come with the BEA TUXEDO system: `CARRAY` (character array possibly containing NULL characters which is neither encoded nor decoded during transmission), `STRING` (NULL-terminated character array), `FML` (and FML32: Fielded Buffers), and `VIEW` (and VIEW32: simple C structures). Note that all views are handled by the same set of routines and that 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 a new buffer type is added or one is deleted, care should be taken 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 source or object file name on the `buildserver`(1) or `buildclient`(1) command line using a `-f` option argument.

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 by the information passed in the TPINIT typed buffer at `tpinit` time and in part by the system based on the processor at 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. `tpcall`'s and `tpforward`'s 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 routine `tpnotify` is used to notify clients identified in this manner.

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 stated here.

```
/*
 * 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#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;

/* 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 transaction id 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 TPSA_FASTPATH            0x00000008     /* System access ==
                                   * fastpath */
 #define TPSA_PROTECTED           0x00000010     /* System access ==
                                   * protected */

/*  /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 */
 long appkey;                     /* application authentication */
                                  /* client key */
 long urcode;                     /* application user-return code */
 CLIENTID cltid;                  /* client identifier for */
                                  /* originating client */
 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 */
 };
 typedef struct tpqctl_t TPQCTL;
```

```
/* /Q structure elements that are valid - set in flags */
 #define TPNOFLAGS                0x00000    /* no flags set -- no get */
 #define TPQCORRID                0x00001    /* set/get correlation id */
 #define TPQFAILUREQ              0x00002    /* set/get failure queue */
 #define TPQBEFOREMSGID           0x00004    /* enqueue before message id */
 #define TPQGETBYMSGID            0x00008    /* dequeue by msgid */
 #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 TPQGETBYCORRID           0x00800    /* dequeue by corrid */

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

/* tperrno values - error codes */
 * The man 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 QMENOTA                   -8
 #define QMEPROTO                  -9
 #define QMEBADQUEUE               -10
 #define QMENOMSG                  -11
 #define QMEINUSE                  -12
 #define QMENOSPACE                -13

/* Event Broker 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 stated here.

```
#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 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.

The table below indicates which functions request/response servers, conversational servers, and clients are allowed to call. Note that `tpsvrinit` and `tpsvrdone` are not in this table since these functions are not called by applications (that is, they are application-supplied functions that are invoked by the BEA TUXEDO system).

Function Call Permissions

Function	**Process Type**		
	Request/response Server	**Conversational Server**	**Client Server**
`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
`tpenqueue`	Y	Y	Y
`tpforward`	Y	N	N
`tpfree`	Y	Y	Y
`tpgetlev`	Y	Y	Y

Function Call Permissions

Function	Process Type		
	Request/response Server	Conversational Server	Client Server
`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
`tpscmt`	Y	Y	Y
`tpsend`	Y	Y	Y
`tpservice`	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

The remaining state tables are for both clients and servers, unless otherwise noted. Keep in mind that because some functions can not 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 client process has been initialized and registered with the transaction manager. Note that this table assumes the use of `tpinit`, which is optional. That is, a client may implicitly join an application by issuing one of many ATMI verbs (for example, `tpconnect` or `tpcall`). A client must use `tpinit` when either application authentication is required (see `tpinit`(3c) and the description of the SECURITY keyword in `ubbconfig`(5)) or the client wishes to directly access an XA-compliant resource manager (see `tpinit`(3c)).

A server is placed in the initialized state by the BEA TUXEDO system's `main` before its `tpsvrinit` function is invoked, and it is placed in the uninitialized state by the BEA TUXEDO 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 process to remain in the same state, unless otherwise noted.

Initialization States

Function	**States**	
	Uninitialized I_0	**Initialized I_1**
`tpalloc`	I_0	I_1
`tpchkauth`	I_0	I_1
`tpfree`	I_0	I_1
`tpinit`	I_1	I_1
`tprealloc`	I_0	I_1
`tpsetunsol`	I_0	I_1
`tpterm`	I_0	I_0
`tptypes`	I_0	I_1
all others (see the following note)	I_1	I_1

Note: all others" refers to the remaining ATMI calls

The remaining state tables assume a precondition of state I (regardless of whether a process arrived in this state via `tpinit` or the BEA TUXEDO system's `main`).

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

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
`tpservice with flag TPTRAN`		R_1
all others	R_0	R_1

The following state table 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 and T assume a precondition of state R (for example, `tpopen` has been called with no subsequent call to `tpclose` or `tpterm`).

Transaction State of Process

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		
`tpterm`	I_0	T_0	
all others	T_0	T_1	T

The following state table indicates the state of a single request descriptor returned by `tpacall`.

Asynchronous Request Descriptor States

Function	States	
	No Descriptor A_0	**Valid Descriptor A_1**
tpacall	A_1	
tpgetrply		A_0
tpcancel		A_0 *
tpabort	A_0	A_0 †
tpcommit	A_0	A_0 †
tpsuspend	A_0	A_1 ‡
tpreturn	A_0	A_0
tpforward	A_0	A_0
tpterm	I_0	I_0
all others	A_0	A_1

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

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

‡ If the descriptor is associated with the caller's transaction, then `tpsuspend` returns a protocol error.

The following state table 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

Connection Request Descriptor States

Function/Event	**States**				
	C_0	**C_1**	**C_2**	**C_3**	**C_4**
`tpconnect with TPSENDONLY`	C_1 *				
`tpconnect with TPRECVONLY`	C_2 *				
`tpservice with flag TPSENDONLY`	C_3 †				
`tpservice with flag TPRECVONLY`	C_4 †				
`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		
`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			

Connection Request Descriptor States

Function/Event	**States**				
	C_0	C_1	C_2	C_3	C_4
`tpterm (client only)`	C_0	C_0			
`tpcommit (originator only)`	C_0	C_0 ‡	C_0 ‡		
`tpsuspend (originator only)`	C_0	C_1 ††	C_2 ††		
`tpabort (originator only)`	C_0	C_0 ‡	C_0 ‡		
`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 others	C_0	C_1	C_2	C_3	C_4

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

† If the `TPTRAN` flag is set, the connection is in transaction mode.

‡ If the connection is not in transaction mode, no state change.

†† If the connection is in transaction mode, then `tpsuspend` returns a protocol error.

TX State Transitions

The BEA TUXEDO system ensures that a process calls the TX verbs 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 primitives are shown in the table below. Calls that return failure do not make state transitions, except where described by specific state table entries. Any BEA TUXEDO system client or server is allowed to use the TX verbs.

The states are defined below:

- S_1: No RMs have been opened or initialized. A process cannot start a global transaction until it has successfully called tx_open.
- S_2: A process has opened its RM but is not in a transaction. Its `transaction_control` characteristic is TX_UNCHAINED.
- S_3: A process has opened its RM but is not in a transaction. Its `transaction_control` characteristic is TX_CHAINED.
- S4: A process has opened its RM and is in a transaction. Its `transaction_control` characteristic is TX_UNCHAINED.
- S5: A process has opened its RM and is in a transaction. Its `transaction_control` characteristic is TX_CHAINED.

Function	**States**				
	S_1	**S_2**	**S_3**	**S_4**	**S_5**
`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
`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

Note: TX_SET1 denotes any of 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 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), `tpservice`(3c), `tpadvertise`(3c), `tpalloc`(3c), `tpbegin`(3c), `tpcall`(3c), `tpconnect`(3c), `tpinit`(3c), `tpopen`(3c), `tuxtypes`(5), `typesw`(5)

AEMsetblockinghook(3)

Name

`AEMsetblockinghook`(3)- establish 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 taks 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 the following condition, `AEMsetblockinghook`() fails and sets `tperrno` to:

`[TPEPROTO]`
: `AEMsetblockinghook`() was called while a blocking operation is in progress.

Portability

This interface is supported only in Mac clients.

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

AEOaddtypesw(3)

Name AEOaddtypesw(3)-install or replace 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 system. For new buffer types, make sure that the WSH(1) and other BEA TUXEDO 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 system’s defined buffer type routines. The application and the BEA TUXEDO system’s buffer routines can be intermixed in one user defined buffer type.

Return Values AEOaddtypesw() returns the number of user buffer types in the system on success. AEOaddtypesw() returns -1 on error and sets tperrno to indicate the error condition.

Errors Under the following condition, AEOaddtypesw() fails and sets tperrno to:

[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 system typeswitch DLL. Please refer to the *BEA TUXEDO Administrator’s Guide* 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 `buffer`(3), `buildwsh`(1), `typesw`(5)

AEPisblocked(3)

Name	`AEPisblocked`-determine 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	`AEPisblocked`() returns 1 if there is an outstanding blocking function awaiting completion. 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 re-entered, depending on the message(s) it receives. In this instance, the `AEPisblocked`() function can be used to ascertain whether the task has been re-entered while waiting for an outstanding blocking call to complete. Note that ATMI prohibits more than one outstanding call per thread.
See Also	`AEPsetblockinghook`()

AEPsetblockinghook(3)

Name `AEPsetblockinghook`-establish an application-specific blocking hook function

Synopsis

```
#include <atmi.h>
int _TM_FARPROC far pascal AEPsetblockinghook(_TM_FARPROC)
```

Description `AEPsetblockinghook()` is an “ATMI Extension for OS/2 Presentation Manager” that allows a OS/2 PM task to install a new function which the ATMI networking software uses to implement blocking ATMI calls. It taks a pointer to the function address of the blocking function to be installed.

A default function, by which blocking ATMI calls are handled, is included. The function `AEPsetblockinghook()` 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())
                ;
}
```

The default BlockingHook() function is equivalent to:

```
BOOL far pascal
win_default(void)
{
        QMSG    qmsg;
        HAB     hab;
        BOOL    ret;

        /* get the next message if any */
        hab = WinQueryAnchorBlock(HWND_DESKTOP);
        if (ret = WinPeekMsg(hab, qmsg, NULL, 0, 0, PM_REMOVE)) {
                /* if we got one, process it */
                WinDispatchMsg(hab, qmsg);
        }
        /* TRUE if we got a message */
        return(ret);
}
```

The `AEPsetblockinghook`() function is provided to support those applications which require more complex message processing - for example, those employing the MDI (multiple document interface) model. It is not intended as a mechanism for performing general application functions. In particular, no ATMI functions may be issued from a custom blocking hook function.

Return Values

`AEPsetblockinghook`() returns a pointer to the function address of the previously installed blocking function. The application or library that calls the `AEPsetblockinghook`() 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 `AEPsetblockinghook`() and eventually use `AEPsetblockinghook`(NULL) to restore the default mechanism.) `AEPsetblockinghook`() returns NULL on error and sets `tperrno` to indicate the error condition.

Errors

Under the following condition, `AEPsetblockinghook`() fails and sets `tperrno` to:

`[TPEPROTO]`
: `AEPsetblockinghook`() was called while a blocking operation is in progress.

Portability

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

Notices

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

See Also

`AEPisblocked`()

AEWaddtypesw(3)

Name

`AEWaddtypesw`-install or replace a user defined buffer type at execution time

Synopsis

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

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

Description

`AEWaddtypesw`() is an “ATMI Extension for Windows” that allows a Windows task 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 BEA TUXEDO system. For new buffer types, make sure that the WSH(1) and other BEA TUXEDO system processes involved in the call processing have been built with the new buffer type.

The function pointers in the `TMTYPESW` array should be obtained by using the MakeProcInstance() function, and these functions should appear in the Module Definition file of the applications in the `EXPORTS` section.

The application can also use the BEA TUXEDO system’s defined buffer type routines like `_dfltinitbuf`(), etc. The application and the BEA TUXEDO system’s buffer routines can be intermixed in one user defined buffer type.

Return Values

`AEWaddtypesw`() returns the number of user buffer types in the system on success. `AEWaddtypesw`() returns -1 on error and sets `tperrno` to indicate the error condition.

Errors

Under the following condition, `AEWaddtypesw`() fails and sets `tperrno` to:

`[TPEINVAL]`
: `AEWaddtypesw`() 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 system typeswitch DLL. Please refer to the *BEA TUXEDO Administrators Guide* for more information.

Notices In the Windows 3.x 16 bit environment, the buffer type information is reset after `tpterm`(3) is called by the application. FAR PASCAL is used only for the 16 bit Windows 3.x environment.

Examples

```
#include <windows.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
};
        lpFinit = MakeProcInstance(Nfinit, hInst);
        lpFreinit = MakeProcInstance(Nfreinit, hInst);
        lpFuninit = MakeProcInstance(Nfuninit, hInst);

        newtype.initbuf = lpFinit;
        newtype.reinitbuf = lpFreinit;
        newtype.uninitbuf = lpFuninit;

        if(AEWaddtypesw(newtype) == -1) {
                userlog("AEWaddtypesw 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 Microsoft Windows'

EXETYPE      WINDOWS


EXPORTS
       Nfinit
       Nfreinit
       Nfuninit
        ....
```

See Also `buffer`(3), `buildwsh`(1), `typesw`(5)

AEWisblocked(3)

Name

`AEWisblocked`-determine if a blocking call is in progress

Synopsis

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

Description

`AEWisblocked`() is an “ATMI Extension for Windows” that allows a Windows task to determine if it is executing while waiting for a previous blocking call to complete.

Return Values

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

Errors

No errors are returned.

Portability

This interface is supported only in DOS Windows clients.

Comments

Although a blocking ATMI call appears to an application as though it “blocks,” the Windows 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 re-entered, depending on the message(s) it receives. In this instance, the AEWisblocked() function can be used to ascertain whether the task has been re-entered while waiting for an outstanding blocking call to complete. Note that ATMI prohibits more than one outstanding call per thread.

See Also

`AEWsetblockinghook`()

AEWsetblockinghook(3)

Name `AEWsetblockinghook`- establish an application-specific blocking hook function

Synopsis
```
#include <atmi.h>
int FARPROC far pascal AEWsetblockinghook(FARPROC)
```

Description `AEWsetblockinghook`() is an "ATMI Extension for Windows" that allows a Windows 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 `AEWsetblockinghook`() 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())
                ;
}
```

The default BlockingHook() function is equivalent to:

```
BOOL far pascal
win_default(void)
{
        MSG     msg;
        BOOL    ret;
        /* get the next message if any */
        if (ret = PeekMessage(msg, NULL, 0, 0, PM_REMOVE)) {
                /* if we got one, process it */
                TranslateMessage(msg);
                DispatchMessage(msg);
        }
        /* TRUE if we got a message */
        return(ret);
}
```

The `AEWsetblockinghook`() function is provided to support those applications which require more complex message processing-for example, those employing the MDI (multiple document interface) model. It is not intended as a mechanism for performing general application functions. In particular, no ATMI functions may be issued from a custom blocking hook function. Note that the blocking hook function should return 0 to terminate the loop and non-zero to continue looping.

Return Values `AEWsetblockinghook`() returns a pointer to the procedure-instance of the previously installed blocking function. The application or library that calls the `AEWsetblockinghook`() 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 `AEWsetblockinghook`() and eventually use `AEWsetblockinghook`(NULL) to restore the default mechanism.) `AEWsetblockinghook`() returns NULL on error and sets `tperrno` to indicate the error condition.

Errors Under the following condition, `AEWsetblockinghook`() fails and sets `tperrno` to:

`[TPEPROTO]`
: `AEWsetblockinghook`() was called while a blocking operation is in progress.

Portability This interface is supported only in DOS Windows clients.

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

See Also `AEWisblocked`()

AEWsetunsol(3)

Name
`AEWsetunsol`-post Windows message for TUXEDO 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 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 TUXEDO unsolicited message arrives, a Windows message is posted. `lParam` is set to the BEA TUXEDO system buffer pointer, or zero if none. If `lParam` is non-zero, the application must call `tpfree`(3) to release the buffer.

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

Return Values
`AEWsetunsol`() returns \-1 on failure and sets `tperrno` to indicate the error condition.

Errors
Under the following conditions, `AEWsetunsol`() fails and sets `tperrno` to:

`[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`(3)

buffer(3c)

Name buffer(3c)-semantics of elements in tmtype_sw_t

Synopsis

```
int     /* Initialize a new data buffer */
_tminitbuf(char *ptr, long len)
int    /* Re-initialize a re-allocated data buffer */
_tmreinitbuf(char *ptr, long len)
int    /* Un-initialize 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)
```

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 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 wild card 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 sub-types. If all sub-types are to be treated the same for a given type, then the wild card character, “*”, can be used. Note that the function tptypes can be used to determine a buffer's type and sub-type if sub-types need to be distinguished. If some subset of the sub-types 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 sub-type values should appear in the switch before the sub-type designated with the wild card. Thus, searching for types and sub-types in the switch is done from top to bottom, and the wild card sub-type entry accepts any "leftover" type matches.

`dfltsize` is used when allocating or re-allocating a buffer. The larger of `dfltsize` and the routines' *`size`* parameter is used to create or re-allocate 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 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 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 sub-types, then the writer of `_tminitbuf` can use `tptypes` to determine the sub-type.

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 re-initialize a re-allocated buffer. It is called from within `tprealloc` after the buffer has been re-allocated.

If no buffer re-initialization 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.

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, pre-processing 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 pre-processing 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.

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 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 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 system where the encoded version of the data will be copied. The un-encoded 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 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 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 Event Broker 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 Event Broker performs the subscription's notification action. Otherwise, if `_tmfilter` returns 0, the Event Broker 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 Event Broker server to convert a buffer's data into a printable string, based on a format specification named *`fmt`*. The Event Broker 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`*.

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)

catgets(3)

Name

`catgets`-read 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`(3). *`s`* points to a default message string which will be returned by `catgets` if the identified message catalogue is not currently available.

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`(3).

catopen(3)

Name

`catopen`, `catclose`-open/close 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 (e.g., in `/etc/profile`) and thus makes the location and naming conventions associated with message catalogues transparent to both programs and users.

The full set of metacharacters is:

Metacharacters

`%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`(3) 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`*.

Diagnostics

If successful, `catopen` returns a message catalogue descriptor for use on subsequent calls to `catgets`(3) and `catclose`(3). Otherwise `catopen`() returns `(nl_catd) -1`. `catclose` returns 0 if successful, otherwise -1.

See Also

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

change_atts(3)

Name `change_atts`-change field attributes on form

Synopsis

```
#include <fml.h>
int change_atts(fbfr,fldid,occno,atts)
FBFR *fbfr;
FLDID fldid;
int occno;
char *atts;
```

Description `change_atts` is a function called by a server to alter dynamically field attributes on a form displayed by a data entry program. `change_atts`() adds a special field to *`fbfr`*, which is interpreted by a data entry program upon receiving the fielded buffer. *`fldid`* and *`occno`* specify the field on the form whose attributes are to change. If two fields on the form have identical *`fldid`* and *`occno`*, both will change. *`atts`* should point to a string of attributes. The available attributes are those allowed in the `flags` field of a UFORM script, with the exception of the `H` and `I` attributes, which are not allowed. Literal fields may not be altered to become protected or unprotected fields, and protected and unprotected fields may not be altered to become literal fields. It is not necessary for *`atts`* to point to a complete list of attributes. Only those attributes which are to change need be included. For example, a field that is described as secret and unprotected on the UFORM script, can be changed to secret and protected with a `P` as its *`atts`* argument. *`atts`* may also point to the string `RESTORE`, in which case all of the original attributes specified by the UFORM script are restored, and the dynamic attributes are forgotten.

Servers in which `change_atts`() is called must link in `libtfrm.a` with the `-f` option of buildserver(1).

Examples The following changes a field from secret and bold to non-secret and non-bold.

```
change_atts(fbfr, fldid, occno, “N0”);
```

Any code that uses `change_atts`() must link in `libtfrm.a`. The following example shows how `libtfrm.a` should be specified on a `buildserver`(1) command line.

```
buildserver -s PRTFORM -f ${TUXDIR}/lib/formprint.o -f lib/libtfrm.a
```

Diagnostics `change_atts`() returns a 1 on success. It has two return codes to indicate failure. It returns a zero on a failed fielded buffer operation. In this case, `Ferror` contains the reason for failure. It returns a \-1 on all other failures.

See Also `buildserver`(1), `compilation`(5), *TUXEDO Data Entry System Guide*

decimal(3)

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 are provided as part of the CICS instantiation of the /Host Extension. The functions allow storage, conversion, and manipulation of packed decimal data on the BEA TUXEDO system. Note that the format in which the decimal data type is represented on the BEA TUXEDO system is different from its representation under CICS.

Native Decimal Representation

Decimals are represented on native BEA TUXEDO 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.

do_form(3)

Name — `do_form`-form display subroutine

Synopsis

```
#include “fml.h”

FBFR *
do_form(formname, fbfr)
char *formname;
FBFR **fbfr;
```

Description — `do_form`() displays *`formname`*, collects input from a user, and returns a pointer to a fielded buffer containing the information entered on a form. If the form was exited with the abort function key, or by pressing the break key, then NULL is returned. On a system error, (`FBFR` *)-1 is returned. *`formname`* should be a file output by `mc`(1). If *`formname`* begins with a slash (`/`) the given path is searched; otherwise, *`formname`* is searched for in the directories listed in the `MASKPATH` environment variable. *`formname`* should include the `.M` file extension. When `do_form`() is called, *`fbfr`* is either a pointer to a pointer to a fielded buffer, a pointer to NULL, or a NULL pointer. If it is a pointer to NULL or a NULL pointer, `do_form`() allocates the fielded buffer. If it is not NULL, information contained in the fielded buffer is displayed on the screen. Upon return, the value contained in *`fbfr`*, if it is not a NULL pointer, points to a fielded buffer containing the screen content. If the value returned by the function is not a NULL and not a -1, then it points to the same fielded buffer. It is the caller’s responsibility to free the fielded buffer pointed to by *`fbfr`* by calling `tpfree`(), regardless of the return value of the function. `do_form`() calls `formexit`() on disastrous conditions. A default version of `formexit`() exists in `$TUXDIR/lib/libtfrm.a`. `do_form` uses `tpalloc`(3) to allocate a buffer and `tpfree`(3) must be used to free the fielded buffer.

Application-defined function keys can be used (including re-mapping the default command and control keys) by exporting the file name in the UDFK environment variable. The file format is described in `udfk`(5).

Examples — This example displays the form supplied in a command line argument and writes the resulting fielded buffer on the standard output.

```
main(argc,argv)
int argc; char *argv[];
{
    FBFR *fbfr, *fbfr1;
    fbfr = (FBFR *)NULL;
    fbfr1 = do_form(argv[1],fbfr);
    if (fbfr1 == (FBFR *)NULL)
        fprintf(stderr, “user quit\en”);
    else if (fbfr1 == (FBFR *)-1)
```

```
                fprintf(stderr,\0”system error\en”);
            else
                Fprint(fbfr1);
            tpfree(fbfr);
        }
```

Diagnostics

If the form was exited with a transmit-form key (i.e., when a service would be called in `mio`(1)), a pointer to a fielded buffer is returned. If the form was exited with an abort function key, or with the break key, NULL is returned and the *`fbfr`* argument contains the pointer to the fielded buffer (if it is not a NULL pointer). On errors, such as `malloc`(3) failures, or failure to read a file, a (`FBFR` *)-1 is returned.

Notices

The form displayed allows full shell escapes.

When compiling, use

```
buildclient -o outputfile -f “appfiles” -l -ltfrm -l -lcurses -l -lm
```

where *`outputfile`* is the executable name, and *`appfiles`* are application files needed.

CAVEAT

`do_form`() is not designed to work with menu hierarchies, specifically calling services from within the hierarchy. When a transmit-form key is entered from a form, `do_form`() returns the associated fielded buffer. If the form is not a top-level form, `do_form`() pops all levels of forms and returns. Data is not propagated up the menu hierarchy, and the current state (the position within the menu hierarchy) is lost.

See Also

`mio`(1), `malloc`(3) in a UNIX System reference manual, *`TUXEDO Data Entry System Guide`*, *`TUXEDO FML Guide`*

formprint(3)

Name `formprint`-print a form

Synopsis

```
#include “fml.h”
extern int LINES;
extern int COLS;
formprint(frmname,fbfr,cmd)
char *frmname;
FBFR *fbfr;
char *cmd;
form1print(frmname,fbfr,file,formfeed, lines, pages)
char *frmname;
FBFR *fbfr;
FILE *file;
char *formfeed;
int *lines;
int *pages;
form2print(frmname,fbfr,buffer,formfeed, lines, pages)
char *frmname;
FBFR *fbfr;
char *buffer;
char *formfeed;
int *lines;
int *pages;
```

Description The `formprint` routines accept the name of a form, *`frmname`*, and a fielded buffer, *`fbfr`*, and replace field areas on the form with the contents of the fielded buffer. The resulting form is output in a format suitable for printing. The default value for `LINES` is 66; for `COLS` it is 132. The routines differ, in that each directs its output to another medium. All three routines have *`frmname`* and *`fbfr`* as common parameters. *`frmname`* should be the name of a standard UFORM form, without the `.M` suffix. If *`frmname`* is null, the name of the form is assumed to be in the reserved FORMNAM field in the fielded buffer.

`formprint`() places its output in a temporary file, and then executes *`cmd`* on that file. `%s` should be substituted for the temporary file name wherever the temporary file name would appear in the *`cmd`* string. If *`cmd`* is null, `lp %s` is assumed to be the command string. If the USPOOLDIR environment variable is set, the temporary file is created in the $USPOOLDIR directory, otherwise the temporary file is created in `/tmp`.

`form1print`() places its output in *`file`*. The *`formfeed`* string is output at the end of each page. Upon successful return, *`page`* is set to the number of pages output, and *`lines`* is set to the number of lines on the last page. The number of pages output is the same as the number of pages on the form.

`form2print`() is identical to `form1print`(), except instead of placing its output in *`file`*, it places its output in *`buffer`*. *`buffer`* should be large enough to handle any anticipated (and unanticipated) output.

Examples `formprint(NULL,fbfr,"cat %s >/dev/tty")` is an acceptable invocation of `formprint`. It sends the form named in the reserved FORMNAM field of the fielded buffer to `/dev/tty`.

Diagnostics These routines return 1 on success and \-1 on failure.

Notices It is not possible to link these routines and the `curses`(3) library (`libcurses.a`) into one program.

See Also `FRMPRT`(5), `curses`(3X) in a UNIX System reference manual

frmmisc(3)

Name

`frmmisc`-miscellaneous forms routines

Synopsis

```
#include “fml.h”

extern char *extmskpath; /* maskpath */
extern char *extcache;   /* mask cache */

int frmval(frmname,fbfr,fldid,oc,errmsg)
char *frmname;           /* form name, without the .M suffix */
FBFR *fbfr;              /* fielded buffer to be validated */
FLDID *fldid;            /* field id of field in error */
int *oc;                 /* occurrence number of field in error */
char **errmsg;           /* error message for incorrect field */

int frmflds(frmname,fldids,occs,max)
char *frmname;           /* form name, without the .M suffix */
FLDID *fldids;           /* points to array of field ids */
int *occs;               /* points to array of occurrences */
int max;                 /* size of fldids and occs arrays */
```

Description

`frmval`() validates a fielded buffer, *`fbfr`*, based on the validations present in the compiled mask *`frmname`*. It returns 1 if *`fbfr`* passes the validation, \-1 if *`frmname`* is non-existent or can’t be read in for any reason, and 0 if *`fbfr`* fails the validations. In the last case *`fldid`* and *`occno`* point to the field id and occurrence number of the field in error. *`errmsg`* points to a character array that contains the error message that would appear on the form’s status line if the form were actually displayed on the screen. The value pointed to by *`errmsg`* is valid only until the next call of `frmval`().

`frmflds`() returns the number of fields present in *`frmname`* and places the field ids and occurrence numbers of those fields in arrays *`fldids`* and *`occs`* respectively. Only *`max`* fields are placed in the arrays, however the actual number of fields on the mask is always returned. `frmflds`() returns a \-1 if it couldn’t access *`frmname`* for any reason.

Prior to calling these routines `extmskpath` should be set to the mask path, and `extcache` should be set to the mask cache address (see loadfiles(1)). When these routines are called from within a validation function that is linked into mio(1) it is not necessary to initialize these variables because they are initialized by `mio`. For the routines listed above, *`frmname`* should be passed as the form name without the `.M` suffix.

Programs callling these functions should be linked with the following libraries in the given order:

```
$TUXDIR/lib/libtfrm.a,
$TUXDIR/lib/libfml.a,
$TUXDIR/lib/libgp.a,
and the standard math library.
```

Notices The callers of these routines may want to supply their own version of `formexit`, a routine that is called in fatal situations, such as `malloc` failures.

See Also `loadfiles`(1), `mio`(1)

gp_mktime(3)

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

Local timezone 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.

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]);
```

See Also ctime(3C), getenv(3C), timezone(4)

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 later 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, if not set, is GMT0. The format for TZ is the following.
`stdoffset[dst[offset],[start[time],end[time]]]`

std and *dst*

Three or more bytes that are the designation for the standard (*std*) and daylight savings time (*dst*) timezones. Only *std* is required, if *dst* is missing, then daylight savings time does not apply in this locale. Upper- and lower-case 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 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 timezone 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*

The *d*th day, (0 *d* 6) of week *n* of month *m* of 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 the *d*th day occurs. Day 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 given is 02:00:00.

maskprt(3)

Name `maskprt`-send mask to FRMPRT server

Synopsis

```
maskprt(fbfr)
FBFR \(**fbfr;
```

Description The function `maskprt`() is used to print a fielded buffer according to a form definition. It could be used, for example, to get a hard copy of the form. `maskprt`() sends the formatted buffer to the BEA TUXEDO system supplied server called `FRMPRT`(5). The buffer must be of type `FML`, and must be obtained by a call to tpalloc(3). `FRMPRT`() accepts the buffer, prints it into a UNIX text file, then calls a command to output the file.

`maskprt`() calls `tpacall`(3) to send the message to `FRMPRT`(5). It fails [`TPNOENT`] if `FRMPRT`(5) is not an active server.

Example

```
maskprt(xxxbuf);
```

See Also `FRMPRT`(5), `tpalloc`(3), `tpcall`(3)

mods(3)

Name mods-modified mask field routines

Synopsis

```
#include “fml.h”
#include “mods.h”

get_mods(fbfr,mod_array,size_mod_array)
FBFR *fbfr;
struct track_mods *mod_array;
int size_mod_array;

mods_needed(fbfr)
FBFR *fbfr;

set_mods(fbfr,fldid,occno,cmd)
FBFR *fbfr;
FLDID fldid;
int occno;
char *cmd;

int fld_mod(fbfr, fldid, occno)
FBFR *fbfr;
FLDID fldid;
int occno;
```

Description The mods routines are used by servers communicating with mio(1) to determine which mask fields have been modified. All the routines described below, have *fbfr* as their first argument. *fbfr* is the fielded buffer returned to a server by mio.

get_mods() places the fldid and occurrence numbers of fields that have been modified on a mio mask into an array of structures, *mod_array*, supplied by the caller. Only *size_mod_array* entries will be made in *mod_array*. mods_needed() should be called to determine the actual *size_mod_array* needed to hold all modified field entries. Once a field has been changed on a mask, it will exist in the list of modified fields until one of the following three things happens: a new mask is displayed, the modified field is reset with a call to set_mods(), or the user clears the entire screen with the clear screen function key. get_mods() returns a -1 on an error, and a non-negative number indicating the number of entries placed in *mod_array* on success. When an error indication is returned, Ferror contains the reason for the error.

`mods_needed`() returns the number of entries needed in *`mod_array`* to hold all modified field information returned by `get_mods`(). It returns a -1 on an internal failure, in which case `Ferror` contains the reason for failure. The value returned by `mods_needed`() may be passed directly to `get_mods`(). If `get_mods`() finds a -1 in its *`size_mod_array`* parameter it will also return a 0.

`set_mods`() sets the modified status of all fields on an `mio` mask with field identifier *`fldid`* and occurence number *`occno`* based on *`cmd`*. *`cmd`* may be either of the strings `“MOD_SET”` or `“MOD_RESET”`, enclosed in quotation marks as shown. If *`cmd`* is `“MOD_RESET”` the indicated fields are not returned in the modified list until they are changed again. If *`cmd`* is `“MOD_SET”` the indicated fields always appear in the modified list, until one of the three conditions listed under the `get_mods`() routine is met. If *`fldid`* is zero then *`cmd`* applies to all protected and unprotected fields on the mask. `set_mods`() returns a 0 on an invalid *`cmd`*, a -1 on an FML error, in which case the reason for the error is in `Ferror`, and a 1 on success. If the `Ferror` is `FNOSPACE` the caller should Frealloc(3) the fielded buffer and try again.

`fld_mod`() returns a 1 if a field specified by *`fldid`* and *`occno`* was modified. It returns a 0 if the specified field was not modified, and a -1 on an internal error. The internal error is usually due to a failed `malloc`(3).

Servers in which `mods` routines are called must link in `libtfrm.a` with the `-f` option of buildserver(1).

Notices Only modifications to fields done through the standard input are tracked. Modifications from other sources, such as asynchronous updates, are not tracked.

See Also `buildserver`(1), `Frealloc`(3), *TUXEDO FML Guide*

nl_langinfo(3)

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);
```

would return a pointer to the string "`Dim`" if the identified language was French and a French locale was correctly installed; or "`Sun`" if the identified language was English.

Diagnostics — If `setlocale`(3) has not been called successfully, or if `langinfo`(5) 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`(3), `strftime`(3), `langinfo`(5), `nl_types`(5).

recomp(3)

Name

recomp, rematch-regular expression compile/execute

Synopsis

```
char *recomp( pattern-1, [pattern-2, ...], 0 )
char *pattern-1, [*pattern-2, ...];

extern int   _Cerrnbr;
extern char *_Cerrmsg[];

char *rematch( pat, text, [substr-0, ..., substr-9,] 0 );
char *pat, *text, [*substr-0, ..., *substr-9];

extern char  *_Mbegin;
extern int    _Merrnbr;
extern char  *_Merrmsg[];
extern char   _Eol;
```

Description

The routines, recomp() and rematch(), provide a regular expression pattern matching scheme for C. There are two parts: a pattern compiler, recomp(); and a pattern interpreter, rematch(). They are, in effect and in spirit, extensions of the standard routines, regcmp(3) and regex(3)

Significant features are the inclusion of regular expression alternation and portability of the code.

recomp() compiles a pattern, in the form of a regular expression, into an intermediate code sequence. rematch() then searches user text for a pattern match by interpreting the codes.

The code sequence, an array of characters, can be computed off-line by the command rex(1), which reads regular expressions from the standard input and writes the corresponding character arrays to the standard output. The output can then be included in a regular C compile.

Regular Expressions

The patterns for these routines are given with regular expressions, 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.

Regular Expressions

Rule	Matching Text
character	itself (*character* is any ASCII character except the special ones mentioned below).
`\ character`	itself except as follows: ♦ \\ -- newline ♦ \\t -- tab ♦ \\b -- backspace ♦ \\r -- carriage return ♦ \\f -- formfeed
`\ special-character`	its *un*special self. The special characters are `. * + ? | ( ) [ {` and `\\`. . -- any character except the end-of-line character (usually newline or null). ^ -- beginning of the line. $ -- 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)
`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 thru 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.

recomp - Regular Expression Compiler

`recomp()` concatenates its arguments up to a terminating zero into a single expression. The expression is then compiled into a character array whose address is returned as the function value.

Space for the array is obtained from the standard C routine, `malloc`(3), and may be released (by the user) with a call to the standard `free`(3) routine.

`recomp()` returns a zero (NULL) value if the pattern cannot be processed. The reason is indicated by a global variable, `_Cerrnbr`, which is set to a non-zero value on any failure. `_Cerrnbr` may be used directly or as an index into a table of error messages, `_Cerrmsg`. `_Cerrnbr` is reset on each call to `recomp()`. The possible values for `_Cerrnbr` and the corresponding messages from `_Cerrmsg` are given below.

Regular Expression Compiler

`_Cerrnbr`	`_Cerrmsg[_Cerrnbr]`
0	"Ok"
1	"Syntax error at col *colnbr*, char '*char*'" (*colnbr* is the position where the error is discovered; *char* is the character at that position)
2	"Out of node storage"
3	"Out of vector storage"
4	"Too many OR's"
5	"More than 255 repetitions" (a number in the "*rE*{...}" construct is greater than 255)
6	"Negative range" (a range for a character class or a closure is given backward)
7	"Out of heap storage" (`malloc` failed)

Conditions that cause `_Cerrnbr` values of 2, 3, and 4 relate to the size of `recomp`()'s internal data structures and are unlikely to occur.

The first and second characters of the code array form the least significant byte and the most significant byte, respectively, of an unsigned 16 bit quantity that gives the length, in bytes, of the entire array. This value will prove useful for copying or otherwise manipulating the array.

rematch -- Regular Expression Matcher

`rematch`() interprets the code sequence produced by `recomp`() to search a user string for a match. When a match is found, `rematch`() returns as its value the address of the first character beyond the matching text (which may then be used as the text argument in a subsequent call to `rematch`()). Also, the variable `_Mbegin` is set to the address of the first character of the matching text.

Any text matching a specified sub-pattern (see "`( rE ) $ n`" above) is copied into the corresponding user buffer, providing one was supplied on the call. All supplied user buffers are reset on each `rematch`() call and filled only on a successful match.

Note: `rematch`(), unlike its role model, `regex`(3), requires a zero terminating argument.

`rematch`() returns NULL if no match can be found or if something else goes wrong. If no match is found the variable, `_Merrnbr`, is set to zero. If something worse happens it is set to a non-zero value. As above, `_Merrnbr` serves as an index for a table of diagnostic messages as indicated below.

`_Merrnbr`	`_Merrmsg[_Merrnbr]`
`0`	"Ok" (If `rematch`() returned NULL, no match was found)
`1`	"Too many closures"
`2`	"Line too long"
`3`	"Corrupt vector" (check `recomp`() for failure)
`4`	"More than 10 substr args" (User probably forgot to terminate `rematch`() arguments with a zero)
`5`	"Too many assignments"

`_Merrnbr` values of 1, 2, or 5 are not likely to occur. They relate to the size of data structures used by `rematch`().

The variable `_Eol` is the current end-of-line character. It is initialized to “`\0`” but may be changed by the user to other reasonable values (e.g., “`\n`”). The end-of-line character determines what the special character, `$`, matches.

Example The following program scans its input for C identifiers and prints each one on a separate line.

```
#include <stdio.h>
main()
{
   char *recomp(), *rematch();
   char *patVect, *cursor, line[100], usrBuf[100];

   patVect = recomp( "([a-zA-Z_][a-zA-Z0-9_]*)$0", 0 );

   while ( gets(line) ) {
       cursor = line;
       while ( cursor=rematch(patVect,cursor,usrBuf,0) )
           printf( "%sn", usrBuf );
   }
}
```

Note the use of the variable, `cursor`, to indicate a successful match as well as to provide (on success) the starting point for the next search. A less courageous programmer would check `recomp`()'s return value and restrict the length of the pattern match to the receiving buffer's size (e.g., "{0,98}" instead of "*").

Implementation `recomp`() and `rematch`() are written in portable C code. `recomp`() employs YACC, which accounts for the fact that it is bigger and somewhat slower than its counterpart, `regcmp`(3). The intermediate code produced by `recomp`() is generally more compact than that of `regcmp`(3).

`rematch`() is about the same size and has about the same speed as its counterpart, `regex`(3).

Notices Support for the functions described in this manual page will be withdrawn in Release 5.0 of the BEA TUXEDO system.

See Also `rex`(1), `ed`(1) in a UNIX System reference manual, `regcmp`(3), `malloc`(3), `free`(3), `regex`(3) in a UNIX System reference manual

rpc_sm_allocate(3)

Name

`rpc_sm_allocate`, `rpc_ss_allocate`-allocates memory within the RPC stub memory management scheme

Synopsis

```
#include <rpc/rpc.h>
idl_void_p_t rpc_sm_allocate(unsigned32 size, unsigned32 *status)
idl_void_p_t rpc_ss_allocate(unsigned32 size)
```

Description

Applications call `rpc_sm_allocate`(3) to allocate memory within the RPC stub memory management scheme. The input parameter, *`size`*, specifies in bytes, the size of memory to be allocated. Before a call to this routine, the stub memory management environment must have been established. For service code that is called from the server stub, the stub itself normally establishes the necessary environment. When `rpc_sm_allocate` is used by code that is not called from the stub, the application must establish the required memory management environment by calling `rpc_sm_enable_allocate`(3).

Specifically, if the parameters of a server stub include any pointers other than those used for passing parameters by reference or the `[enable_allocate]` attribute is specified for the operation in the ACS file, then the environment is automatically set up. Otherwise, the environment must be set up by the application by calling `rpc_sm_enable_allocate`.

When the stub establishes the memory management environment, the stub itself frees any memory allocated by `rpc_sm_allocate`. The application can free such memory before returning to the calling stub by calling `rpc_sm_free`(3).

When the application establishes the memory management environment, it must free any memory allocated, either by calling `rpc_sm_free` or by calling `rpc_sm_disable_allocate`(3).

The output parameter, *`status`*, returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not. Possible status codes and their meanings include:

`rpc_s_ok`
: Always returned. The return value is used to determine failure.

`rpc_ss_allocate` is the exception-returning version of this function and has no *`status`* output parameter. No exceptions are raised.

Return Values

On success, the routines return a pointer to the allocated memory. Note that in the ISO standard C environments, `idl_void_p_t` is defined as `void *` and in other environments is defined as `char *`. Insufficient memory is reported by returing a NULL pointer.

See Also

`rpc_sm_free`(3), `rpc_sm_enable_allocate`(3), `rpc_sm_disable_allocate`(3) , *BEA TUXEDO TxRPC Guide*

rpc_sm_client_free(3)

Name
: `rpc_sm_client_free`, `rpc_ss_client_free`-frees memory returned from a client stub

Synopsis

```
#include <rpc/rpc.h>
void rpc_sm_client_free (idl_void_p_t node_to_free, unsigned32 *status)
void rpc_ss_client_free (idl_void_p_t node_to_free)
```

Description
: The `rpc_sm_client_free` routine releases memory allocated and returned from a client stub. The input parameter, *`node_to_free`*, specifies a pointer to memory returned from a client stub. Note that in the ISO standard C environments, `idl_void_p_t` is defined as `void *` and in other environments is defined as `char *`.

This routine enables a routine to deallocate dynamically allocated memory returned by an RPC call without knowledge of the memory management environment from which it was called.

Note that this routine is always called from client code, even if the code can is executing as part of a server.

The output parameter, *`status`*, returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not. Possible status codes and their meanings include:

`rpc_s_ok`
: Success.

`rpc_ss_client_free` is the exception-returning version of this function and has no *`status`* output parameter. No exceptions are raised.

Return Values
: None.

See Also
: `rpc_sm_free`(3), `rpc_sm_set_client_alloc_free`(3), `rpc_sm_swap_client_alloc_free`(3), *TUXEDO TxRPC Guide*

rpc_sm_disable_allocate(3)

Name

rpc_sm_disable_allocate, rpc_sm_disable_allocate-releases resources and allocated memory within the stub memory management scheme

Synopsis

```
#include <rpc/rpc.h>
void rpc_sm_disable_allocate(unsigned32 *status);
void rpc_ss_disable_allocate(void);
```

Description

The `rpc_sm_disable_allocate` routine releases all resources acquired by a call to `rpc_sm_enable_allocate`(3), and any memory allocated by calls to `rpc_sm_allocate`(3) after the call to `rpc_sm_enable_allocate` was made.

The `rpc_sm_enable_allocate` and `rpc_sm_disable_allocate` routines must be used in matching pairs. Calling this routine without a previous matching call to `rpc_sm_enable_allocate` results in unpredictable behavior.

The output parameter, *`status`*, returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not. Possible status codes and their meanings include:

`rpc_s_ok`
  Success.

`rpc_ss_disable_allocate` is the exception-returning version of this function and has no *`status`* output parameter. No exceptions are raised.

Return Values

None.

See Also

`rpc_sm_allocate`(3), `rpc_sm_enable_allocate`(3), *BEA TUXEDO TxRPC Guide*

rpc_sm_enable_allocate(3)

Name `rpc_sm_enable_allocate`, `rpc_ss_enable_allocate`-enables the stub memory management environment

Synopsis
```
#include <rpc/rpc.h>
void rpc_sm_enable_allocate(unsigned32 *status)
void rpc_ss_enable_allocate(void)
```

Description Applications can call `rpc_sm_enable_allocate` to establish a stub memory management environment in cases where one is not established by the stub itself. A stub memory management environment must be established before any calls are made to `rpc_sm_allocate`(3). For service code called from the server stub, the stub memory management environment is normally established by the stub itself. Code that is called from other contexts needs to call `rpc_sm_enable_allocate` before calling `rpc_sm_allocate` (e.g., if the service code is called directly instead of from the stub).

The output parameter, *`status`*, returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not. Possible status codes and their meanings include:

`rpc_s_ok`
    Success.

`rpc_s_no_memory`
    Insufficient memory available to set up necessary data structures.

`rpc_ss_enable_allocate` is the exception-returning version of this function and has no *`status`* output parameter. The following exceptions are raised by this routine.

`rpc_x_no_memory`
    Insufficient memory available to set up necessary data structures.

Return Values None.

See Also `rpc_sm_allocate`(3), `rpc_sm_disable_allocate`(3), *TUXEDO TxRPC Guide*

rpc_sm_free(3)

Name

`rpc_sm_free, rpc_ss_free`-frees memory allocated by the rpc_sm_allocate routine

Synopsis

```
#include <rpc/rpc.h>
void rpc_sm_free(idl_void_p_t node_to_free, unsigned32 *status)
void rpc_ss_free(idl_void_p_t node_to_free)
```

Description

Applications call `rpc_sm_free` to release memory allocated by rpc_sm_allocate(3). The input parameter, *`node_to_free`*, specifies a pointer to memory allocated by `rpc_sm_allocate`. Note that in ISO standard C environments, `idl_void_p_t` is defined as `void *` and in other environments is defined as `char *`.

When the stub allocates memory within the stub memory management environment, service code called from the stub can also use `rpc_sm_free` to release memory allocated by the stub.

Unpredictable behavior results if `rpc_ss_free` is called with a pointer to memory not allocated by `rpc_sm_allocate` or memory allocated by `rpc_sm_allocate`, but not the first address of such an allocation.

The output parameter, *`status`*, returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not. Possible status codes and their meanings include:

`rpc_s_ok`
: Success.

`rpc_ss_free` is the exception-returning version of this function and has no *`status`* output parameter. No exceptions are raised.

Return Values

None.

See Also

`rpc_sm_allocate`(3), *`TUXEDO TxRPC Guide`*

rpc_sm_set_client_alloc_free(3)

Name rpc_sm_set_client_alloc_free, rpc_ss_set_client_alloc_free-sets the memory allocation and freeing mechanisms used by the client stubs

Synopsis
```
#include <rpc/rpc.h>
void rpc_sm_set_client_alloc_free(
   idl_void_p_t (*p_allocate)(unsigned long size),
   void (*p_free) (idl_void_p_t ptr), unsigned32 *status)

void rpc_ss_set_client_alloc_free(
   idl_void_p_t (*p_allocate)(unsigned long size),
   void (*p_free) (idl_void_p_t ptr))
```

Description The rpc_sm_set_client_alloc_free routine overrides the default routines that the client stub uses to manage memory. The input parameters, *p_allocate* and *p_free* specify memory allocator and free routines. The default memory management routines are ISO C malloc() and free() except when the remote call occurs within server code in which case the memory management routines must be rpc_ss_allocate(3) and rpc_ss_free(3).

The output parameter, *status*, returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not. Possible status codes and their meanings include:

rpc_s_ok
    Success.

rpc_s_no_memory
    Insufficient memory available to set up necessary data structures.

rpc_ss_set_client_alloc_free is the exception-returning version of this function and has no *status* output parameter. The following exceptions are raised by this routine.

rpc_x_no_memory
    Insufficient memory available to set up necessary data structures.

Return Values None.

See Also rpc_sm_allocate(3), rpc_sm_free(3), *BEA TUXEDO TxRPC Guide*

rpc_sm_swap_client_alloc_free(3)

Name
`rpc_sm_swap_client_alloc_free`, `rpc_ss_swap_client_alloc_free`-exchanges current memory allocation and freeing mechanism used by client stubs with one supplied by client

Synopsis
```
#include <rpc/rpc.h>
void rpc_sm_swap_client_alloc_free(
  idl_void_p_t (*p_allocate)(unsigned long size),
  void (*p_free) (idl_void_p_t ptr),
  idl_void_p_t (**p_p_old_allocate)(unsigned long size),
  void (**p_p_old_free)( idl_void_p_t ptr),
  unsigned32 *status)

void rpc_ss_swap_client_alloc_free(
  idl_void_p_t (*p_allocate)(unsigned long size),
  void (*p_free) (idl_void_p_t ptr),
  idl_void_p_t (**p_p_old_allocate)(unsigned long size),
  void (**p_p_old_free)( idl_void_p_t ptr))
```

Description
The `rpc_sm_swap_client_alloc_free` routine exchanges the current allocate and free mechanisms used by the client stubs for routines supplied by the caller. The input parameters, *`p_allocate`* and *`p_free`*, specify new memory allocation and free routines. The output parameters, *`p_p_old_allocate`* and *`p_p_old_free`* return the memory allocation and free routines in use before the call to this routine.

When a callable routine is an RPC client, it may need to ensure which allocate and free routines are used, despite the mechanism its caller had selected. This routine allows scoped replacement of the allocation/free mechanism to allow this.

The output parameter, *`status`*, returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not. Possible status codes and their meanings include:

`rpc_s_ok`
Success.

`rpc_s_no_memory`
Insufficient memory available to set up necessary data structures.

`rpc_ss_swap_client_alloc_free` is the exception-returning version of this function and has no *`status`* output parameter. The following exceptions are raised by this routine.

`rpc_x_no_memory`
: Insufficient memory available to set up necessary data structures.

Return Values None.

See Also `rpc_sm_allocate`(3), `rpc_sm_free`(3), `rpc_sm_set_client_alloc_free`(3), *`BEA TUXEDO system Guide`*

setlocale(3)

Name

`setlocale`-modify and query 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 theBEA TUXEDO 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.

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 `ctime`(3C), `ctype`(3C), `getdate`(3C), `localeconv`(3C), `printf`(3S), `strftime`(3C), `strtod`(3C), `environ`(5), `mklanginfo`(1)

strerror(3)

Name	`strerror`-get 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.
See Also	`perror`(3)

strftime(3)

Name `strftime`-convert 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`*.

Directives

%%	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.

Directives

`%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 (e.g. 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.

Selecting the Output Language By default, the output of `strftime`, appears in US 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`(3).

Timezone The timezone 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`(3)

tpabort(3)

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`(3), 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`(3) 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`(3) or with the `TPNOTRAN` flag (that is, not in transaction mode) are not affected.

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

Return Values `tpabort`() returns \-1 on error and sets `tperrno` to indicate the error condition.

Errors Under the following conditions, `tpabort`() fails and sets `tperrno` to:

[`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 can have been heuristically completed.

[`TPEPROTO`]
: `tpabort`() 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.

[TPEOS]
An operating system error has occurred.

Notices When using `tpbegin`(3), `tpcommit`(3) and `tpabort`() to delineate a BEA TUXEDO 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`(3) or `tpabort`().

See Also `tpbegin`(3), `tpcommit`(3), `tpgetlev`(3)

tpacall(3)

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 `tpsprio`(3). If *`data`* is non-NULL, it must point to a buffer previously allocated by `tpalloc`(3) 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 sub-type of *`data`* must match one of the types and sub-types recognized by *`svc`*. Note that for each request sent while in transaction mode, a corresponding reply must ultimately be received.

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 re-issued. tpacall() fails and

Return Values Upon successful completion, `tpacall`() returns a descriptor that can be used to receive the reply of the request sent. Otherwise it returns a value of \-1 and sets `tperrno` to indicate the error condition.

Errors Under the following conditions, `tpacall`() fails and 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`(3), or *`flags`* are invalid).

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

`[TPEITYPE]`
: The type and sub-type of *`data`* is not one of the allowed types and sub-types 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]`
: A timeout occurred. If the caller is in transaction mode, then a transaction timeout occurred and the transaction is marked abort-only; otherwise, a blocking timeout occurred and neither `TPNOBLOCK` nor `TPNOTIME` was specified. If a transaction timeout 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).

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

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

[TPEPROTO]
: tpacall() was called in an improper context.

[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(3), tpcall(3), tpcancel(3), tpgetrply(3), tpgprio(3), tpsprio(3)

tpadmcall(3)

Name tpadmcall-administer 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 verb 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`(3) 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`(3). 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`(5).

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`(5) and `APPQ_MIB`(5) 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`(5) for which they have the appropriate permissions. Similarly, the caller may GET and SET any attributes for any class in `APPQ_MIB`(5), 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`(5) for which they have the appropriate permissions. Additionally, the caller may GET and SET any attributes for any class in `APPQ_MIB`(5), subject to class-specific restrictions.

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

Environment Variables

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

`TUXCONFIG`

> File or device name where 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

Under the following conditions, `tpadmcall`() fails and sets `tperrno` to one of the following values. 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 in an improper context.

[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(3).

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 Release 5.0 or later.

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

See Also MIB(5), TM_MIB(5), APPQ_MIB(5), EVENT_MIB(5), ACL_MIB(5), WS_MIB(5), *BEA TUXEDO Administrator’s Guide*

tpadvertise(3)

Name

`tpadvertise`(3)-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 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

`tpadvertise` returns -1 on error and sets `tperrno` to indicate the error condition.

Errors

Under the following conditions, `tpadvertise` fails and sets `tperrno` to:

`[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(3)

Name `tpalloc`(3)-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 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 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`().

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 Under the following conditions, `tpalloc`() fails and sets `tperrno` to:

`[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 in an improper context.

[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`(3c), `realloc`(3c), or `free`(3c) 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 system extension. Details are in `intro`(3c).

See Also

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

tpbegin(3)

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 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`(3) or `tpabort`(3) 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`(3) or `tpabort`(3). Participants can influence the outcome of a transaction by the return values (*`rval`*s) they use when they call `tpreturn`(3). 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`(3) 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 time-out 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.

Return Values `tpbegin`() returns \-1 on error and sets `tperrno` to indicate the error condition.

Errors

Under the following conditions, tpbegin() fails and sets tperrno to:

[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(3), and tpabort(3) to delineate a BEA TUXEDO 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(3) or tpabort(3). 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 system transaction.

See Also

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

tpbroadcast(3)

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`(3). *`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.

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.

Return Values `tpbroadcast`() returns \-1 on failure and sets `tperrno` to indicate the error condition.

Errors Under the following conditions, `tpbroadcast`() fails, sends no broadcast messages to application clients, and sets `tperrno` to:

`[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 and neither `TPNOBLOCK` nor `TPNOTIME` was 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 in an improper context.

`[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`(3) 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`(5) for a detailed discussion of notification methods.)

Note that signaling of clients is always done by the system so that the behavior of notification is consistent regardless of where the originating notification call is made. Because of this, only clients running as the application administrator can use signal-based notification. The id for the application administrator is identified as part of the configuration file 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`(3) for more information on notification method selection.

See Also `tpalloc`(3), `tpinit`(3), `tpnotify`(3), `tpterm`(3), `ubbconfig`(5)

tpcall(3)

Name

`tpcall`(3)-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`(3c) immediately followed by `tpgetrply`(3c). `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 `tpsprio`(3c). The data portion of a request is pointed to by *`idata`*, a buffer previously allocated by `tpalloc`(3c). *`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 sub-type of *`idata`* must match one of the types and sub-types 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.

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 `intro`(3c) 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.

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 sub-type of the received buffer must match the type and sub-type 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 re-issued.

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`(3c). `tpcall` returns -1 on error and sets `tperrno` to indicate the error condition. If a call fails with a particular `tperrno` value, a subsequent call to `tperrordetail`(3c) 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

Under the following conditions, `tpcall` fails and 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]`
: Can not 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 sub-type of *`idata`* is not one of the allowed types and sub-types that *`svc`* accepts.

`[TPEOTYPE]`
: Either the type and sub-type of the reply are not known to the caller; or, `TPNOCHANGE` was set in *`flags`* and the type and sub-type of **`odata`* do not match the type and sub-type 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]`
: A timeout occurred. If the caller is in transaction mode, then a transaction timeout occurred and the transaction is marked abort-only; otherwise, a blocking timeout occurred and neither `TPNOBLOCK` nor `TPNOTIME` was specified. In either case, neither **`odata`*, its contents, nor **`olen`* is changed. If a transaction timeout 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).

[TPESVCFAIL]
: The service routine sending the caller's reply called tpreturn(3c) 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 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).

[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 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). 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 in an improper context.

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

tpcancel(3)

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`(3). 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.

Return Values `tpcancel`() returns \-1 on error and sets `tperrno` to indicate the error condition.

Errors Under the following conditions, `tpcancel`() fails and sets `tperrno` to:

`[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 in an improper context.

`[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`(3)

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`(3c) to determine if a password should be obtained from the user.

Return Values — `tpchkauth`() returns one of the following non-negative values on success.

`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.

It returns -1 on error and sets `tperrno` to indicate the error condition.

Errors — Under the following conditions, `tpchkauth`() fails and sets `tperrno` to:

`[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. 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`(3c) message is generated and the method is automatically set to dip-in.

See Also — `tpinit`(3c)

tpchkunsol(3)

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 system libraries.

Return Values

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

Errors

Under the following conditions, `tpchkunsol`() fails and sets `tperrno` to:

`[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`(3) 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.) Note that signaling of clients is always done by the system so that the behavior of notification is consistent regardless of where the originating notification call is made. Because of this, only clients running as the application administrator can use signal-based notification. The ID for the application administrator is identified as part of the configuration file 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`(3) for more information on notification method selection.

See Also `tpbroadcast`(3), `tpinit`(3), `tpnotify`(3), `tpsetunsol`(3)

tpclose(3)

namSe 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.

Return Values `tpclose`() returns \-1 on error and sets `tperrno` to indicate the error condition.

Errors Under the following conditions, `tpclose`() fails and sets `tperrno` to:

`[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`(3)

tpcommit(3)

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`(3) 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`(3)), `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`(3) or with the `TPNOTRAN` flag (that is, connections not in transaction mode) are not affected by calls to `tpcommit()` or `tpabort`(3).

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

Return Values

`tpcommit()` returns \-1 on error and sets `tperrno` to indicate the error condition.

Errors

Under the following conditions, `tpcommit()` fails and sets `tperrno` to:

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

`[TPETIME]`
: The transaction timed out and the status of the transaction is unknown (that is, it can have been either committed or aborted). Note that if the transaction timed out and its status is known to be aborted, then `TPEABORT` is returned.

`[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.

`[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 can have been heuristically completed.

`[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.

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

Notices When using `tpbegin`(), `tpcommit`() and `tpabort`() to delineate a BEA TUXEDO 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 system transaction.

See Also `tpabort`(3), `tpbegin`(3), `tpconnect`(3), `tpgetlev`(3), `tpreturn`(3), `tpscmt`(3)

tpconnect(3)

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`(3). *`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 sub-type of *`data`* must match one of the types and sub-types 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`(3) to get the data.

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 re-issued.

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 Under the following conditions, `tpconnect`() fails and sets `tperrno` to an error code listed below. (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`* is non-NULL and does not point to a buffer allocated by `tpalloc`(3), `TPSENDONLY` or `TPRECVONLY` was not specified in *`flags`*, or *`flags`* are otherwise invalid).

`[TPENOENT]`
: Cannot initiate a connection to *`svc`* because it does not exist or is not 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 connections has been reached.

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

[TPETIME]
: A timeout occurred. If the caller is in transaction mode, then a transaction timeout occurred and the transaction is marked abort-only; otherwise, a blocking timeout occurred and neither TPNOBLOCK nor TPNOTIME were specified. If a transaction timeout occurred, then any attempts to send or receive messages on any connections or to start a new connection will fail with TPETIME until the transaction has been aborted.

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

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

[TPEPROTO]
: tpconnect() was called in an improper context.

[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(3), tpdiscon(3), tprecv(3), tpsend(3), tpservice(3)

tpconvert(3c)

Name
`tpconvert`-convert 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.

Return Values
`tpconvert`() returns -1 on failure 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(3).

Portability

This interface is available only on BEA TUXEDO Release 5.0 or later. This interface is available on workstation platforms.

See Also

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

tpcryptpw(3)

Name
: `tpcryptpw`-encrypt application password in 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.

Return Values
: `tpcryptpw`() returns -1 on failure and sets `tperrno` to indicate the error condition.

Errors
: Under the following conditions, `tpcryptpw`() fails and 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(3).

Portability
: This interface is available only on UNIX System sites running BEA TUXEDO 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), BEA TUXEDO Administrator's Guide

tpdequeue(3)

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`() dequeues 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 default 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 using the *`ctl`* parameter. *`ctl`* flags can also be used to indicate that the application wants to wait for a message, in the case where a message is not currently available. 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`(3). 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 deleted 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 deletion of the message from the queue is also rolled back). This can be exploited to "peek" at a message on the queue, rolling back the transaction to leave the message on the queue (note that this cannot be done in `TPNOTRAN` mode as described below). 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. The message is dequeued in a separate transaction. If a communication error or a timeout occurs (either transaction or blocking timeout), the application will not know whether or not the message was successfully dequeued and the message may be lost.

Following is a list of valid *`flags`*.

`TPNOTRAN`
: If the caller is in transaction mode and this flag is set, then the message is not dequeued within the same transaction as the caller. 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 (for example, the internal buffers into which the message is transferred are full). 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). This blocking condition does not include blocking on the queue itself if the TPQWAIT option is specified.

`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.

`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 sub-type of the dequeued message must match the type and sub-type of the buffer pointed to by **`data`*.

`TPSIGRSTRT`
: If a signal interrupts any underlying system calls, then the interrupted system call is re-issued. When `TPSIGRSTRT` is not specified and a signal interrupts a system call, then `tpdequeue`() fails and `tperrno` is set to `TPGOTSIG`.

If `tpdequeue`() returns successfully, the application can retrieve additional information about the message using *`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 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. This is described below.

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 */
```

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`
: If set, it requests that the message identified by *ctl->msgid* be dequeued. The message identifier would be one that was returned by a prior call to `tpenqueue`(3). Note that the message identifier is not valid if the message has moved from one queue to another; in this case, use the correlation identifier. This option cannot be used with the `TPQWAIT` option.

`TPQGETBYCORRID`
: If set, it requests that the message with the correlation identifier specified by *ctl->corrid* be dequeued. The correlation identifier would be one that the application specified when enqueuing the message with `tpenqueue`(). This option cannot be used with the `TPQWAIT` option.

`TPQWAIT`
: If set, it indicates that an error should not be returned if the queue is empty. Instead, the process should block until a message is available.

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 */
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 */
```

Following is a list of valid bits for the *flags* parameter controlling output information from `tpdequeue`(). If the flag bit is turned on when `tpdequeue`() is called, then the associated element in the structure is populated if available and the bit remains set. If the value is not available, the flag bit will be turned off after `tpdequeue`() completes.

`TPQPRIORITY`
: If set and the value is available, the priority at which the message was queued 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).

`TPQMSGID`
: If set and the call to `tpdequeue`() was successful, the message identifier will be stored in *ctl->msgid*.

`TPQCORRID`
: If set and the call to `tpdequeue`() was successful and the message was queued with a correlation identifier, the value will be stored in *ctl->corrid*. Any reply to a queue must have this correlation identifier.

`TPQREPLYQ`
: If set and the message is associated with a reply queue, the value will be 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 set and the message is associated with a failure queue, the value will be stored in *ctl->failurequeue*. Any failure message should go to the named failure queue within the same queue space as the request message.

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, *ctl->appkey* is set to 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 This function returns \-1 on error and sets `tperrno` to indicate the error condition.

Errors Under the following conditions, `tpdequeue()` fails and sets `tperrno` to one of the following (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(3)` or *`flags`* are invalid).

`[TPENOENT]`
: Cannot access the *`qspace`* because it is not available (the associated `TMQUEUE(5)` server is not available).

`[TPEOTYPE]`
: Either the type and sub-type of the dequeued message are not known to the caller; or, `TPNOCHANGE` was set in *`flags`* and the type and sub-type of **`data`* do not match the type and sub-type of the dequeued message. Regardless, neither **`data`*, its contents nor **`len`* are changed. When this error occurs, the transaction is marked abort-only and the message will remain on the queue.

`[TPETIME]`
: A timeout occurred. If the caller is in transaction mode, then a transaction timeout occurred and the transaction is to be aborted; otherwise, a blocking timeout occurred and neither `TPNOBLOCK` nor `TPNOTIME` were specified. If a transaction timeout occurred, any attempts to dequeue new messages will fail with `TPETIME` until the transaction has been aborted.

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

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

`[TPEPROTO]`
: `tpdequeue()` was called in an improper context. 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 made with the TPNOTRAN flag and an error occurred trying to start a transaction in which to dequeue the message.

`[QMEBADMSGID]`
: An invalid message identifier was specified for dequeuing.

`[QMEINUSE]`
: When dequeuing a message by correlation or message identifier, the specified message is in-use by another transaction. Otherwise, all messages currently on the queue are in-use by other transactions.

`[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.

See Also `TMQUEUE`(5), `tpalloc`(3), `tpenqueue`(3)

tpdiscon(3)

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`(3) 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`(3); 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
`tpdiscon`() function returns \-1 on error and sets `tperrno` to indicate the error condition.

Errors
Under the following conditions, `tpdiscon`() fails and sets `tperrno` to:

`[TPEBADDESC]`
`cd` is invalid or is the descriptor with which a conversational service was invoked.

`[TPETIME]`
A timeout occurred. The descriptor is no longer valid.

`[TPEPROTO]`
tpdiscon() was called in an improper context.

`[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`(3), `tpcommit`(3), `tpconnect`(3), `tprecv`(3), `tpreturn`(3), `tpsend`(3)

tpenqueue(3)

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 system server, the *`qname`* matches the name of a service provided by a 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 expected a reply, then the reply to the forwarded service request is stored on the originator's (stable) queue. 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 system processes (clients and/or servers). In this case, the queue name does not match a service name but some agreed upon title for transferring the message.

If *`data`* is non-NULL, it must point to a buffer previously allocated by `tpalloc`(3) 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`(3).

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 deleted 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. In this case, the queued message is stored on the queue in a separate transaction. Once `tpenqueue`() returns successfully, the submitted message is guaranteed to be available. If a communication error or a timeout occurs (either transaction or blocking timeout), 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.

Following is a list of valid *`flags`*.

`TPNOTRAN`
: If the caller is in transaction mode and this flag is set, then the message is not queued within the same transaction as the caller. 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 (for example, the internal buffers into which the message is transferred are full). 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 re-issued. When `TPSIGRSTRT` is not specified and a signal interrupts a system call, then `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; 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.

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 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 bit 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 bit 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.

`TPQTIME_ABS`
: If 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`() or `mktime`() (the number of seconds since 00:00:00 UTC, January 1, 1970). `TPQTIME_ABS` and `TPQTIME_REL` are mutually exclusive flags.

`TPQTIME_REL`

If set, the message is made available after a time relative to the completion of the queuing transaction. *ctl->deq_time* specifies the number of seconds to delay after the transaction completes before the submitted message should be available. `TPQTIME_ABS` and `TPQTIME_REL` are mutually exclusive flags.

`TPQPRIORITY`

If 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).

`TPQCORRID`

If set, the correlation identifier value specified in \%*ctl->corrid* is available when a message is dequeued with `tpdequeue`(3). This identifier accompanies any reply or failure message that is queued such that an application can correlate a reply with a particular request. The entire value should be initialized (e.g., padded with null characters) such that the value can be matched at a later time.

`TPQREPLYQ`

If 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 set, a failure queue named in *ctl->failurequeue* is associated with the queued message. If a failure occurs when the enqueued message is subsequently dequeued, a failure message will go 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).

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 */
```

Following is a list of valid bits for the *`flags`* parameter controlling output information from `tpenqueue`(). If the flag bit is turned on when `tpenqueue`() is called, then the associated element in the structure is populated if available and the bit remains set. If the value is not available, the flag bit will be turned off after `tpenqueue`() completes.

`TPQMSGID`
: If set and the call to `tpenqueue`() was successful, the message identifier will be stored in *`ctl->msgid`*.

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 This function returns \-1 on error and sets `tperrno` to indicate the error condition. Otherwise, the message has been successfully queued when `tpenqueue`() returns.

Errors Under the following conditions, `tpenqueue`() fails and sets `tperrno` to 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`(3), or *`flags`* are invalid).

`[TPENOENT]`
: Cannot access the *`qspace`* because it is not available (the associated `TMQUEUE`(5) server is not available).

`[TPETIME]`
: A timeout occurred. If the caller is in transaction mode, then a transaction timeout occurred and the transaction is to be aborted; otherwise, a blocking timeout occurred and neither `TPNOBLOCK` nor `TPNOTIME` was specified. If a transaction timeout occurred, any attempts to enqueue new messages will fail with `TPETIME` until the transaction has been aborted.

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

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

`[TPEPROTO]`
: `tpenqueue`() was called in an improper context.

`[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 made with the TPNOTRAN flag and an error occurred trying to start a transaction in which to enqueue the message.

`[QMEBADMSGID]`
: An invalid message identifier was specified.

`[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]`
: An enqueue was done when the transaction state was not active.

`[QMEBADQUEUE]`
: An invalid or deleted queue name was specified.

`[QMENOSPACE]`
: There is no space on the queue for the message.

See Also `TMQFORWARD`(5), `TMQUEUE`(5), `gp_mktime`(3), `tpalloc`(3), `tpacall`(3), `tpinit`(3), `tpsprio`(3)

tperrordetail(3c)

Name
`tperrordetail`(3c)-get additional detail about an error generated from the last BEA TUXEDO 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 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 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.

Return Values
`tperrordetail` returns a -1 on error 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_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`.

`TPED_TERM`
: A Workstation client has been disconnected from the application.

`TPED_NOUNSOLHANDLER`
: A client does not have an unsolicited handler set. The `TPACK` flag is used in a `tpnotify`(3c) call and the target of the `tpnotify`(3c) is in a BEA TUXEDO session, but it has not set an unsolicited notification handler. When `tpnotify`(3c) fails, `tperrno` is set to `TPENOENT`. A subsequent call to `tperrordetail`(3c) with no intermediate ATMI calls returns `TPED_NOUNSOLHANDLER`.

`TPED_NOCLIENT`
: No client exists. The `TPACK` flag is used in a `tpnotify` call but there is no target for `tpnotify`(3c). When `tpnotify`(3c) fails, `tperrno` is set to `TPENOENT`. A subsequent call to `tperrordetail`(3c) with no intermediate ATMI calls returns `TPED_NOCLIENT`.

TPED_CLIENTDISCONNECTED
A Jolt client is disconnected currently. The TPACK flag is used in a tpnotify(3c) call and the target of tpnotify(3c) is a currently disconnected Jolt client. When tpnotify(3c) fails, a call to tperrordetail(3c) with no intermediate ATMI calls returns TPED_CLIENTDISCONNECTED.

TPED_DOMAINUNREACHABLE
A domain is unreachable. Specifically, a domain that has been configured to satisfy a request that a local domain cannot service, was not reachable when a request was made. If, after the request failure, a call is made to tperrordetail(3c) with no intermediate ATMI calls, TPED_DOMAINUNREACHABLE is returned.
When calls to tpcall(3c), tpgetrply(3c), and tprecv(3c) fail because of an unreachable domain, TPED_DOMAINUNREACHABLE is returned. The following table indicates the corresponding values returned by tperrno.

ATMI Call	**tperrno**	**Error Detail**
tpcall	TPESVCERR	TPED_DOMAINUNREACHABLE
tpgetrply	TPESVCERR	TPED_DOMAINUNREACHABLE
tprecv	TPEEVENT TPEV_SVCERR	TPED_DOMAINUNREACHABLE

Note: 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.

Errors
Under the following conditions tperrordetail fails and sets tperrno to the following:

TPEINVAL
flags not set to zero

See Also
intro(3c), tpstrerrordetail(3c), tperrno(5)

tpforward(3)

Name

`tpforward`(3)-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`(3) in that it is the last call made in a service routine. Like `tpreturn`(3), `tpforward` should be called from within the service routine dispatched to ensure correct return of control to the BEA TUXEDO 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`(3) or `tpabort`(3). If a transaction was explicitly started with `tpbegin`(3) while in a service routine, the transaction must be ended with either `tpcommit`(3) or `tpabort`(3) 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`(3). 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`(3). If this is the same buffer passed to the service routine upon its invocation, then its disposition is up to the BEA TUXEDO 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).

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 while in the service routine or while forwarding the request, the requester waiting for a reply with either `tpcall`(3), or `tpgetrply`(3) will get a `TPETIME` error return. Also, the waiting requester will not receive any data. Service routines, however, are expected to terminate using either `tpreturn`(3) or `tpforward`. A conversational service routine must use `tpreturn`(3), and cannot use `tpforward`.

If a service routine returns without using either `tpreturn`(3) or `tpforward` (that is, 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`(3) or `tpforward` are used outside of a service routine (for example, in clients, or in `tpsvrinit`(3) or `tpsvrdone`(3)), then these routines simply return having no effect.

See Also

`tpalloc`(3), `tpconnect`(3), `tpreturn`(3), `tpservice`(3), `tpstrerrordetail`(3c)

tpfree(3)

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`(3) or `tprealloc`(3). 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 system routine or used in any other manner.

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`(3C;), `realloc`(3C;) or `free`(3C;) in the C library (for example, a buffer allocated with `tpalloc`(3) should not be freed with `free`(3C)).

See Also

`intro`(3), `tpalloc`(3), `tprealloc`(3)

tpgetadmkey(3)

Name

`tpgetadmkey`-get 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 (i.e., *`tpinfo->cltname`*) of either `tpsysadm` or `tpsysop`; otherwise, a valid administrative   key will not be returned.

Return Values

A non-0 value with the high-order bit (0x80000000) set is returned on success; otherwise 0 is returned. 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), `tpinit`(3), `AUTHSVR`(5), BEA TUXEDO Administrator's Guide

tpgetlev(3)

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.

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; otherwise, `tpgetlev`() returns \-1 on error and sets `tperrno` to indicate the error condition.

Errors

Under the following conditions, `tpgetlev`() fails and sets `tperrno` to:

`[TPEPROTO]`
: tpgetlev() was called in an improper context.

`[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`(3), `tpcommit`(3) and `tpabort`(3) to delineate a BEA TUXEDO 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`(3) or `tpabort`(3). 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 system transaction.

See Also

`tpabort`(3), `tpbegin`(3), `tpcommit`(3), `tpscmt`(3)

tpgetrply(3)

Name
`tpgetrply`(3c)-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`(3c) returns a reply from a previously sent request. This function's first argument, *`cd`*, points to a call descriptor returned by `tpacall`(3c). 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`(3c) and *`len`* should point to a long that `tpgetrply`(3c) 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.

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 `intro`(3c) 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.

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 sub-type of the received buffer must match the type and sub-type 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 re-issued.

Except as noted below, **cd* is no longer valid after its reply is received.

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`. `tpgetrply` returns -1 on error and sets `tperrno` to indicate the error condition.

Errors Under the following conditions, `tpgetrply`(3c) fails and 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`(3c) 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 sub-type of the reply are not known to the caller; or, TPNOCHANGE was set in *flags* and the type and sub-type of **data* do not match the type and sub-type 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]
: A timeout occurred. If the caller is in transaction mode, then a transaction timeout occurred and the transaction is marked abort-only; otherwise, a blocking timeout occurred and neither TPNOBLOCK nor TPNOTIME were specified. In either case, neither **data*, its contents nor **len* are changed. **cd* remains valid unless the caller is in transaction mode (and TPGETANY was not set). If a transaction timeout 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(3c) with TPNOTRAN, TPNOBLOCK and TPNOREPLY set).

[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 so long as the transaction has not timed out, further communication may be performed before completely 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).

[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 so long as the transaction has not timed out, further communication may be performed before completely

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). 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 in an improper context.

[`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(3)

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. Priorities can range from 1 to 100, inclusive, with 100 being the highest priority. `tpgprio`() may be called after `tpcall`(3) or `tpacall`(3), (also `tpenqueue`(3), or `tpdequeue`(3), 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.

Since the conversation primitives are not associated with priorities, issuing `tpsend`(3) or `tprecv`(3) has no affect on the priority returned by `tpgprio`(). Also, there is no priority associated with a conversational service routine unless a `tpcall`(3) or `tpacall`(3) is done within that service.

Return Values Upon success, `tpgprio`() returns a request's priority; otherwise `tpgprio`() returns \-1 on error and sets `tperrno` to indicate the error condition.

Errors Under the following conditions, `tpgprio`() fails and sets `tperrno` to:

`[TPENOENT]`
: `tpgprio`() was called and no requests (via `tpcall`(3) or `tpacall`(3)) have been sent, or it is called within a conversational service for which no requests have been sent.

`[TPEPROTO]`
: `tpgprio`() was called in an improper context.

`[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`(3), `tpcall`(3), `tpdequeue`(3), `tpenqueue`(3), `tpservice`(3), `tpsprio`(3)

tpinit(3)

Name tpinit(3)-routine for joining an application

Synopsis

```
#include <atmi.h>
int tpinit(TPINIT *tpinfo)
```

Description tpinit() allows a client to join a BEA TUXEDO system application. Before a client can use any of the BEA TUXEDO system communication or transaction routines, it must first join a BEA TUXEDO system application. Because calling tpinit() is optional, a client may also join an application by calling many ATMI routines (for example, tpcall(3)) 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 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.

If tpinit() is called more than once (that is, after the client has already joined the application), no action is taken and success is returned.

tpinit()'s argument, *tpinfo*, is a pointer to a typed buffer of type TPINIT and a NULL sub-type. 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(3). The buffer should be freed using tpfree(3) 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;
```

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 /WS clients. Note that grpname is not related to ACL GROUPS.

The setting of flags is used to indicate both the client-specific notification mechanism and the mode of system access. These settings may override the application default; however, in the event that they cannot, tpinit() will print a warning in a log file, ignore the setting and return the application default setting in the flags element upon return from tpinit(). For client notification, the possible values for flags are as follows:

TPU_SIG-Select unsolicited notification by signals.

TPU_DIP-Select unsolicited notification by dip-in.

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.

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 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

`tpinit`() returns -1 on error and sets `tperrno` to indicate the error condition.

Errors

Under the following conditions, `tpinit`() fails and sets `tperrno` to:

[`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.

[`TPEPROTO`]
`tpinit`() 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.

Interoperability

`tpchkauth`(3c) 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`(3c) message is generated and the method is automatically set to dip-in.

Environment Variables

`WSENVFILE`-is 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 more details on environment variable settings necessary for workstation clients. Note that this file is processed only when `tpinit`() is called and not before.

`WSNADDR`-is used within `tpinit`() when invoked by a workstation client. It indicates the network address(es) of the workstation listener that is to be contacted for access to the application.

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 example, 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 would look like this:

```
set WSNADDR=(//m1.acme.com:3050^|//m2.acme.com:3050),//m3.acme.com:3050
```

The carat (^) is needed to escape the pipe (|).

The BEA TUXEDO 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.

`WSDEVICE`-is 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`-is 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`-is 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 varies with each instantiation. The instantiation specific Programmer’s Guide should be consulted for further information.

`TMMINENCRYPTBITS`-When connecting to the BEA TUXEDO system, require at least this minimum level of encryption. “0” means no encryption, while “40” and “128” specify the encryption key length (in bits). If this minimum level of encryption cannot be met, link establishment will fail. The default is “0”.

`TMMAXENCRYPTBITS`-When connecting to the BEA TUXEDO system, negotiate encryption up to this level. “0” means no encryption, while “40” and “128” specify the encryption length (in bits). The default is “128”

Warning Signal restrictions may prevent the system 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 `ubbconfig`(5) description of the `*RESOURCES NOTIFY` parameter for a detailed discussion of notification methods.) Note that signaling of clients is always done by the system so that the behavior of notification is consistent regardless of where the originating notification call is made. Because of this, only clients running as the application administrator can use signal-based notification. 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 `tpterm`(3)

tpnotify(3)

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`(3). *`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.

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.

Return Values

tpnotify() returns -1 on failure and sets tperrno to indicate the error condition. If a call fails with a particular tperrno value, a subsequent call to tperrordetail(3c) 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

Under the following conditions, tpnotify() fails and sets tperrno to:

[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.

[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 in an improper context.

[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 `intro`(3), `tpalloc`(3), `tpbroadcast`(3), `tpchkunsol`(3), `tperrordetail`(3c),`tpinit`(3), `tpsetunsol`(3), `tpstrerrordetail`(3c), `tpterm`(3)

tpopen(3)

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.

Return Values

`tpopen`() returns \-1 on error and sets `tperrno` to indicate the error condition.

Errors

Under the following conditions, `tpopen`() fails and sets `tperrno` to:

`[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`(3)

tppost(3)

Name `tppost`-post 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 system event broker 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 system itself.

If *`data`* is non-NULL, it must point to a buffer previously allocated by `tpalloc`(3) 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 event broker. 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 event broker 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`(3). 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`(3), are also dispatched by the event broker but not as part of the posting process' transaction.

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 event broker 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 re-issued. When `TPSIGRSTRT` is not specified and a signal interrupts a system call, then `tppost` fails and `tperrno` is set to `TPGOTSIG`.

Return Values

Upon successful return from `tppost`, `tpurcode` contains the number of event notifications dispatched by the event broker 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 event broker on behalf of *`eventname`*. This function returns -1 on error and sets `tperrno` to indicate the error condition.

Errors

Under the following conditions, `tppost` fails and 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 system Event Broker.

[TPETRAN]
: The caller is in transaction mode, TPNOTRAN was not set and tppost contacted an event broker that does not support transaction propagation (that is, TMUSREVT(5) is not running in a BEA TUXEDO system group that supports transactions).

[TPETIME]
: A timeout occurred. If the caller is in transaction mode, then a transaction time-out occurred and the transaction is to be aborted; otherwise, a blocking time-out occurred and neither TPNOBLOCK nor TPNOTIME were specified. If a transaction timeout occurred, any attempts to do new work will fail with TPETIME until the transaction has been aborted.

[TPESVCFAIL]
: The event broker 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 event broker 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 in an improper context.

[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`(3), `tpunsubscribe`(3), `EVENTS`(5), `TMUSREVT`(5), `TMSYSEVT`(5)

tprealloc(3)

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`(3), 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 re-allocated. 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`() re-initializes a buffer (in a communication manager-specific manner) after it is re-allocated and before it is returned. Thus, the buffer returned to the caller is ready for use.

Return Values
Upon successful completion, `tprealloc`() 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 error condition.

Errors
If the re-initialization function fails, `tprealloc`() fails returning NULL and the contents of the buffer pointed to by *`ptr`* may not be valid. Under the following conditions, `tprealloc`() fails and sets `tperrno` to:

`[TPEINVAL]`
Invalid arguments were given (for example, *`ptr`* does not point to a buffer originally allocated by `tpalloc`(3)).

`[TPEPROTO]`
`tprealloc`() was called in an improper context.

`[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 re-initialization 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`(3C), `realloc`(3C) or `free`(3C) in the C library (for example, a buffer allocated with `tprealloc`() should not be freed with `free`()).

See Also `tpalloc`(3), `tpfree`(3), `tptypes`(3)

tprecv(3)

Name

`tprecv(3)`-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`(3) or the `TPSVCINFO` parameter to the service. The second argument, *`data`*, is the address of a pointer to a buffer previously allocated by `tpalloc`(3c).

`data` must be the address of a pointer to a buffer previously allocated by `tpalloc`(3c) 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 `intro`(3) 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.

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 sub-type 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(3c), or it issued tpreturn(3c), tpcommit(3c) 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`(3c). `tpreturn`(3c) encountered an error that precluded the service from returning successfully. For example, bad arguments may have been passed to `tpreturn`(3c) or `tpreturn`(3c) 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`(3c) with `TPFAIL` or `TPEXIT`). If the subordinate service was in control of this connection when `tpreturn`(3c) 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`(3c) 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.

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`(3). `tprecv`() returns -1 on error and sets `tperrno` to indicate the error condition. If a call fails with a particular `tperrno` value, a subsequent call to `tperrordetail`(3c) 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

Under the following conditions, `tprecv`() fails and sets tperrno to:

`[TPEINVAL]`
: Invalid arguments were given (for example, data is not the address of a pointer to a buffer allocated by `tpalloc`(3c) 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]`
: A timeout occurred. If the caller is in transaction mode, then a transaction timeout occurred and the transaction is marked abort-only; otherwise, a blocking timeout occurred and neither `TPNOBLOCK` nor `TPNOTIME` were specified. In either case, neither *`*data`* nor its contents are changed. If a transaction timeout occurred, then any attempts to send or receive messages on any connections or to start a new connection will fail with `TPETIME` until the transaction has been aborted.

`[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`. However, if 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`(3c). The return value is available in the global variable *`tpurcode`* and the buffer is available in *`data`*.

See Also `tpalloc`(3), `tpconnect`(3), `tpdiscon`(3), `tperrordetail`(3c), `tpsend`(3), `tpservice`(3), `tpstrerrordetail`(3c)

tpresume(3)

Name

`tpresume`-resume 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`(3), or complete it with one of `tpcommit`(3) or `tpabort`(3) at a later time.

The caller must ensure that its linked resource managers have been opened (via `tpopen`(3)) 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.

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`(3) 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`(3), `tpcommit`(3), `tpopen`(3), `tpsuspend`(3)

tpreturn(3c)

Name
`tpreturn`(3c)-routine for returning from a 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 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 system dispatcher.

`tpreturn` is used to send a service's reply message. If the program receiving the reply is waiting in either `tpcall`(3c), `tpgetrply`(3c), or `tprecv`(3c), 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`(3c) directly. To ensure correct results, the program that connected to the conversational service should not call `tpdiscon`(3c); 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 where 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 nor rolled back until either `tpcommit`(3c) or `tpabort`(3c) 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 `tpreturn` 's arguments. *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 is the same as `TPFAIL`, with respect to completing the service, but the server will exit after the transaction is rolled back and the reply is sent back to the requester. If the server is restartable, then the server will automatically be restarted.

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(3c). If this is the same buffer passed to the service routine upon its invocation, then its disposition is up to the BEA TUXEDO 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 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. 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 system dispatcher; thus, it is declared as a void. Service routines, however, are expected to terminate using either tpreturn or tpforward(3c). A conversational service routine must use tpreturn, and cannot use tpforward(3c). If a service routine returns without using either tpreturn or tpforward(3c) (that is, it uses the C language return statement or just simply "falls out of the function") or tpforward(3c) 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`(3c) are used outside of a service routine (for example, in clients, or in `tpsvrinit`(3c) or `tpsvrdone`(3c)), 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`(3c) or `tpgetrply`(3c), and cause the event, `TPEV_SVCERR`, to be sent over the conversation to a program using `tpsend`(3c) or `tprecv`(3c).

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`(3c) and `tpstrerrordetail`(3c) can be used to get additional information about an error produced by the last BEA TUXEDO 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)

tpscmt(3)

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(3) 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 system thread of control calls tpcommit(3). 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(3). If tpforward(3) is called, then the transaction will be completed by the server ultimately calling tpreturn(3). Thus, the setting of the TP_COMMIT_CONTROL characteristic in the service that calls tpreturn(3) determines when tpcommit(3) returns control within a server. If tpcommit(3) returns a heuristic error code, the server will write a message to a log file.

When a client joins a BEA TUXEDO system application, the initial setting for this characteristic comes from a configuration file. (See the CMTRET variable in the *RESOURCES section of ubbconfig(5))

Following are the valid settings for *flags*.

TP_CMT_LOGGED

This flag indicates that tpcommit(3) 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(3) 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(3) has already returned (although the BEA TUXEDO 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`(3) should return after the two-phase commit protocol has finished completely. This setting allows for `tpcommit`(3) to return an indication that a heuristic decision occurred during the second phase of commit.

Return Values Upon success, `tpscmt`() returns the previous value of the `TP_COMMIT_CONTROL` characteristic; otherwise it returns -1 on error and sets `tperrno` to indicate the error condition.

Errors Under the following conditions, `tpscmt`() fails and sets `tperrno` to:

`[TPEINVAL]`
: *`flags`* is not one of `TP_CMT_LOGGED` or `TP_CMT_COMPLETE`.

`[TPEPROTO]`
: `tpscmt`() was called in an improper context.

`[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`(3), `tpcommit`(3) and `tpabort`(3) to delineate a BEA TUXEDO 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 system transaction.

See Also `tpabort`(3), `tpbegin`(3), `tpcommit`(3), `tpgetlev`(3)

tpsend(3)

Name tpsend(3)-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`(3c) or the `TPSVCINFO` parameter passed to a conversational service.

The second argument, *data*, must point to a buffer previously allocated by `tpalloc`(3c). *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 sub-type of *data* must match one of the types and sub-types recognized by the other end of the connection.

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`(3c) 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 re-issued.

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(3c), or it issued tpreturn(3c), tpcommit(3c) or tpabort(3c) 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(3c) without having control of the conversation. In addition, tpreturn(3c) 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(3c) without having control of the conversation. In addition, tpreturn(3c) 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.

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 Under the following conditions, `tpsend`(3c) fails and sets `tperrno` to:

[`TPEINVAL`]
: Invalid arguments were given (for example, *`data`* does not point to a buffer allocated by `tpalloc`(3c) or *`flags`* are invalid).

[`TPEBADDESC`]
: *`cd`* is invalid.

[`TPETIME`]
: A timeout occurred. If the caller is in transaction mode, then a transaction timeout occurred and the transaction is marked abort-only; otherwise, a blocking timeout occurred and neither `TPNOBLOCK` nor `TPNOTIME` was specified. In either case, no changes are made to **`data`*, its contents nor **`len`*. If a transaction timeout occurred, then any attempts to send or receive messages on any connections or to start a new connection will fail with `TPETIME` until the transaction has been aborted.

[`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(3)

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`(3), `tpacall`(3) or `tpforward`(3) routines as well as by services that communicate via `tpconnect`(3), `tpsend`(3) and `tprecv`(3) routines.

Service routines processing requests made via either `tpcall`(3) or `tpacall`(3) 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`(3)).

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 (i.e., 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`(3), `tpcommit`(3), and `tpabort`(3). Note that `tpreturn`(3) is not used to complete a transaction. Thus, it is an error to call `tpreturn`(3) 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. 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`(3) in the communication manager. This buffer may be grown by the user with `tprealloc`(3); however, it cannot be freed by the user. It is recommended that this buffer be the one passed to either `tpreturn`(3) or `tpforward`(3) 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`(3) or `tpforward`(3). *`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`(3) and `tprecv`(3) 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`(3) or `tpforward`(3). A conversational service routine must use `tpreturn`(3), and cannot use `tpforward`(3). If a service routine returns without using either `tpreturn`(3) or `tpforward`(3) (i.e., it uses the C language `return` statement or just simply “falls out of the function”) or `tpforward`(3) 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`(3) or `tpforward`(3) are used outside of a service routine (e.g., in clients, or in `tpsvrinit`(3) or `tpsvrdone`(3)), then these routines simply return having no effect.

Errors

Since `tpreturn`(3) 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`(3) or `tpgetrply`(3), and cause the event, `TPEV_SVCERR`, to be sent over the conversation to a program using `tpsend`(3) or `tprecv`(3).

See Also

`servopts`(5), `tpalloc`(3), `tpbegin`(3), `tpcall`(3), `tpconnect`(3), `tpforward`(3), `tpreturn`(3)

tpsetunsol(3)

Name `tpsetunsol`-routine for setting the method of 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 system libraries. Before the first call to `tpsetunsol`(), any unsolicited messages received by the BEA TUXEDO 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`(3)).

The function pointer passed on the call to `tpsetunsol`() must conform to the parameter definition given. *`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` can not be freed by application code. However, the system frees it and invalidates the data area following return.

Processing within the application unsolicited message handling routine is restricted to the following BEA TUXEDO system calls: `tpalloc`(3), `tpgetlev`(3), `tprealloc`(3) `tptypes`(3), `tpfree`(3).

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); otherwise, it returns `TPUNSOLERR` and sets `tperrno` to indicate the error condition.

Errors Under the following conditions, `tpsetunsol`() fails and sets `tperrno` to:

[`TPEPROTO`]
: `tpsetunsol`() was called in an improper context (e.g., 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`(3) are supported on native site UNIX-based and Windows NT 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`(3), `tpterm`(3)

tpsprio(3)

Name

`tpsprio`-routine for setting service request priority

Synopsis

```
#include <atmi.h>
int tpsprio(prio, flags)
```

Description

`tpsprio`() sets the priority for the next request sent or forwarded. The priority set affects only the next request sent. (Priority can also be set for messages enqueued or dequeued by `tpenqueue`(3) or `tpdequeue`(3) 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`(3) or `tpsend`(3).

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.

Return Values

`tpsprio`() returns \-1 on error and sets `tperrno` to indicate the error condition.

Errors

Under the following conditions, `tpsprio`() fails and sets `tperrno` to:

`[TPEINVAL]`
: *`flags`* are invalid.

`[TPEPROTO]`
: `tpsprio`() was called in an improper context.

`[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`(3), `tpcall`(3), `tpdequeue`(3), `tpenqueue`(3), `tpgprio`(3)

tpstrerror(3)

Name — `tpstrerror`(3)-get error message string for a BEA TUXEDO 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 system function call returns a `-1` or other failure value.

You can use the pointer returned by `tpstrerror`() as an argument to `userlog`(3c) or the UNIX function `fprintf`(3).

Return Values — If *err* is an invalid error code, `tpstrerror`() returns a NULL. On success, the function returns a pointer to a string that contains the error message text.

Errors — `tpstrerror`() returns a NULL on error, 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 — `Fstrerror`(3), `userlog`(3c)

tpstrerrordetail(3)

Name `tpstrerrordetail`-get error detail message string for a BEA TUXEDO 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 system error. *`err`* is the value returned by `tperrordetail`(3).

The user can use the pointer returned by `tpstrerrordetail` as an argument to `userlog`(3c) or the UNIX function `fprintf`(3).

Currently *`flags`* is reserved for future use and must be set to 0.

Return Values If *`err`* is an invalid error code, `tpstrerrordetail` returns a NULL. On success, the function returns a pointer to a string that contains the error detail message text.

Errors `tpstrerrordetail` returns a NULL on error, 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 `intro`(3c), `tperrordetail`(3c), `tpstrerror`(3c), `userlog`(3c), `tperrno`(5)

tpsubscribe(3c)

Name

`tpsubscribe`-subscribe 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 system Event Broker, `TMUSREVT`(5), and are used to notify subscribers when events are posted via `tppost`(3). 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" (for example, "account_error" and "ERROR_STATE" events would both qualify).

If present, *`filter`* is a string containing a boolean filter rule associated with *`eventexpr`* that must be evaluated successfully before the event broker posts the event. Upon receiving an event to be posted, the event broker applies the filter rule, if one exists, to the posted event's data. If the data passes the filter rule, the event broker invokes the notification method associated with *`eventexpr`*; 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 complier (see `Fboolco`(3) and `Fvboolco`(3), respectively) and evaluated against the posted buffer (see `Fboolev`(3) and `Fvboolev`(3), respectively). For STRING buffers, the filter rule is a regular expression. All other buffer types require customized filter evaluators (see `buffer`(3) 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 system client process and *`ctl`* is NULL, then the event broker 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 event broker 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`(3)) an unsolicited message handling routine. If a BEA TUXEDO 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 event broker's list of active subscriptions before exiting (see `tpunsubscribe`(3) for details). Using `tpunsubscribe`'s wild-card 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 event broker 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;
```

The following is a list of valid bits for the *`ctl->flags`* element controlling options for event subscriptions.

`TPEVSERVICE`

Setting this flag bit indicates that the subscriber wants event notifications to be sent to the BEA TUXEDO system service routine named in *`ctl->name1`*. That is, when an event name is posted that evaluates successfully against *`eventexpr`*, the event broker 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 system service name and it may or may not be active at the time the subscription is made. Service routines invoked by the event broker should return with no reply data. That is, they should call `tpreturn`(3) with a NULL data argument. Any data passed to `tpreturn`(3) will be dropped. TPEVSERVICE and TPEVQUEUE are mutually exclusive flags.

If TPEVTRAN is also set in *`ctl->flags`*, then if the process calling `tppost`(3) is in transaction mode, the event broker calls the subscribed service routine such that it will be part of the poster's transaction. Both the event broker, `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 event broker calls the subscribed service routine such that it will not be part of the poster's transaction.

`TPEVQUEUE`

Setting this flag bit 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 event broker 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 event broker 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 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 event broker'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 the `tpenqueue`(3) manual page (specifically, see the section describing the valid list of flags controlling input information for `tpenqueue`(3)). TPEVSERVICE and TPEVQUEUE are mutually exclusive flags.

If TPEVTRAN is also set in *`ctl->flags`*, then if the process calling `tppost`(3) is in transaction mode, the event broker enqueues the posted event and its data such that it will be part of the poster's transaction. The event broker, `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 event broker enqueues the posted event and its data such that it will not be part of the poster's transaction.

`TPEVTRAN`

Setting this flag bit indicates that the subscriber wants the event notification for this subscription to be included in the poster's transaction, if one exists. If this flag bit 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 system Event Broker deletes subscriptions when the resource to which it is posting is not available (for example, the event broker cannot access a service routine and/or a queue space/queue name associated with an event subscription). Setting this flag bit 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 bit is not used, the event broker 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 bit is used with TPEVTRAN and the resource is not available at the time of event notification, then the event broker 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 event broker'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 event broker'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 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 event broker.

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 event broker.

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 event broker. 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 event broker.

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 re-issued. When `TPSIGRSTRT` is not specified and a signal interrupts a system call, then `tpsubscribe` fails and `tperrno` is set to `TPGOTSIG`.

Return Values

Upon successful completion, `tpsubscribe` returns a handle that can be used to remove this subscription from the event broker's list of active subscriptions. Otherwise the function returns -1 and sets `tperrno` to indicate the error condition. Either the subscriber or any other process is allowed to use the returned handle to delete this subscription.

Errors

Under the following conditions, `tpsubscribe` fails and 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 system Event Broker.

`[TPELIMIT]`
: The subscription failed because the event broker's maximum number of subscriptions has been reached.

`[TPEMATCH]`
: The subscription failed because it matched one already listed with the event broker.

`[TPETIME]`
: A timeout occurred. If the caller is in transaction mode, then a transaction timeout occurred and the transaction is to be aborted; otherwise, a blocking timeout occurred and neither `TPNOBLOCK` nor `TPNOTIME` were specified. If a transaction timeout occurred, any attempts to do new work will fail with `TPETIME` until the transaction has been aborted.

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

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

`[TPEPROTO]`
: `tpsubscribe` was called in an improper context.

`[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`(3), `EVENTS`(5), `EVENT_MIB`(5), `Fboolco`(3), `Fboolev`(3), `Fvboolco`(3), `Fvboolev`(3), `recomp`(3), `TMSYSEVT`(5), `TMUSREVT`(5), `tpenqueue`(3), `tppost`(3), `tpsetunsol`(3), `tpunsubscribe`(3), `tuxtypes`(5), `typesw`(5), `ubbconfig`(5)

tpsuspend(3)

Name

`tpsuspend`-suspend 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`(3) may be suspended with `tpsuspend`(). Either the suspending process or another process may use `tpresume`(3) 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`(3) 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 (i.e., `tprecv`(3) 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.

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`(3), `tpbegin`(3), `tprecv`(3), `tpresume`(3)

tpsvrdone(3c)

`tpsvrdone`(3c)-BEA TUXEDO system server termination routine

Synopsis

```
#include <atmi.h>
void tpsvrdone(void)
```

Description

The BEA TUXEDO 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 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 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 system is called instead. The default `tpsvrdone` calls `tpclose` and `userlog` to announce that the server is about to exit.

Usage

If either `tpreturn`(3c) or `tpforward`(3c) is called in `tpsvrdone`, it simply returns having no effect.

See Also

`servopts`(5), `tpclose`(3c), `tpsvrinit`(3c)

tpsvrinit(3)

Name

`tpsvrinit`(3)-the BEA TUXEDO system server initialization routine

Synopsis

```
#include <atmi.h>
int tpsvrinit(int argc, char **argv)
```

Description

The BEA TUXEDO 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 system communication may be performed and transactions may be defined in this routine. However, if `tpsvrinit`() returns with open connections, asynchronous replies pending or while still in transaction mode, the BEA TUXEDO system will close the connections, ignore replies pending, abort the transaction, and the server will exit gracefully.

If an application does not provide this routine in a server, then the default version provided by the BEA TUXEDO system is called instead. The default `tpsvrinit`() calls `tpopen`() and `userlog`() to announce that the server has successfully started.

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`(3C) is used in a BEA TUXEDO system server abstraction, `optarg`, `optind` and `opterr` may be used to control option parsing and error detection in `tpsvrinit`().

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 should not call `exit`(2) itself.

Return Values

A negative return value will cause the server to exit gracefully.

Usage

If either `tpreturn`() or `tpforward`() are used outside of a service routine (e.g., in clients, or in `tpsvrinit`() or `tpsvrdone`()), then these routines simply return having no effect.

See Also

`getopt`(3C), `servopts`(5), `tpopen`(3), `tpsvrdone`(3)

tpterm(3)

Name
`tpterm`-routine for leaving an application

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

Description
`tpterm`() removes a client from a BEA TUXEDO system application. If the client is in transaction mode, then the transaction is rolled back. When `tpterm`() returns successfully, the caller can no longer communicate with any other program nor can it participate in any transactions. Any outstanding conversations are immediately disconnected.

If `tpterm`() is called more than once (that is, after the caller has already left the application), no action is taken and success is returned.

Return Values
`tpterm`() returns \-1 on error and sets `tperrno` to indicate the error condition.

Errors
Under the following conditions, `tpterm`() fails and sets `tperrno` to:

`[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`(3)

tptypes(3)

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`(3). 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.

Return Values Upon success, `tptypes`() returns the size of the buffer; otherwise it returns \-1 upon failure and sets `tperrno` to indicate the error condition.

Errors Under the following conditions, `tptypes`() fails and sets `tperrno` to:

`[TPEINVAL]`
: Invalid arguments were given (for example, *`ptr`* does not point to a buffer gotten from \% tpalloc(3)).

`[TPEPROTO]`
: `tptypes`() was called in an improper context.

`[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`(3), `tpfree`(3), `tprealloc`(3)

tpunadvertise(3)

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 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.

`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 *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

`tpunadvertise`() returns \-1 on error and sets `tperrno` to indicate the error condition.

Errors

Under the following conditions, `tpunadvertise`() fails and sets `tperrno` to:

`[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`(3)

tpunsubscribe(3)

Name `tpunsubscribe`-unsubscribe 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 TUXEDO System Event Broker's list of active subscriptions. *`subscription`* is an event subscription handle returned by `tpsubscribe`(3). Setting *`subscription`* to the wild-card 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`(3). Persistent subscriptions can be deleted only by using the handle returned by `tpsubscribe`(3).

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 wild-card to unsubscribe to any subscriptions made by the original server).

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 re-issued. When `TPSIGRSTRT` is not specified and a signal interrupts a system call, then `tpunsubscribe` fails and `tperrno` is set to `TPGOTSIG`.

Return Values

Upon completion of `tpunsubscribe`, `tpurcode`() contains the number of subscriptions deleted (zero or greater) from the event broker's list of active subscriptions. `tpurcode` may contain a number greater than 1 only when the wild-card handle, -1, is used. Also, `tpurcode` may contain a number greater than 0 even when `tpunsubscribe` completes unsuccessfully (that is, when the wild-card handle is used, the event broker may have successfully removed some subscriptions before it encountered an error deleting others). `tpunsubscribe` returns -1 on error and sets `tperrno` to indicate the error condition.

Errors

Under the following conditions, `tpunsubscribe` fails and 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 system event broker.

`[TPETIME]`
: A timeout occurred. If the caller is in transaction mode, then a transaction timeout occurred and the transaction is to be aborted; otherwise, a blocking timeout occurred and neither `TPNOBLOCK` nor `TPNOTIME` were specified. If a transaction timeout occurred, any attempts to do new work will fail with `TPETIME` until the transaction has been aborted.

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

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

`[TPEPROTO]`
: `tpunsubscribe` was called in an improper context.

`[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

`EVENTS`(5), `EVENT_MIB`(5), `TMSYSEVT`(5), `TMUSREVT`(5), `tppost`(3), `tpsubscribe`(3)

TRY(3)

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 (e.g., *errno* or status variables passed back from an RPC operation). These macros are defined in texc.h and this header is automatically included in any header files generated by tidl(1).

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(3) 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` (e.g., 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 (e.g., 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 (e.g., 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) 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 (e.g. `goto, break, continue, longjmp`(3)).

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; don't 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 propagate status values associated with operating system signals and exceptions raised by the RPC run-time primitives. For each status value, an exception has been pre-defined (for example, exception `rpc_x_no_memory` is defined for status `rpc_s_no_memory`); these are declared in the trpcsts.h header file. While not necessary (since the status exceptions are pre-defined), a status exception can be declared by the application and initialized with the `exc_set_status`() macro which takes a pointer to the EXCEPTION to be initialized, and the status value. The status value associated with a *`status`* exception can be retrieved using the `exc_get_status`() macro. It takes a pointer to the EXCEPTION and a pointer to the variable in which the status value is to be returned; the value of the macro is 0 if it is a *`status`* exception, and -1 otherwise.

The second 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 (e.g., 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 (e.g., printing the error to the userlog), `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
```

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 following exceptions 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.

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.

Built-In Exceptions

Exception	Description
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	
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 (e.g., 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_" (e.g., 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 system header file is texc.h. It is not possible for the same source file to use both DCE and BEA TUXEDO system exception handling. Further, within a program, the handling of signal exceptions can only be done by either DCE or the BEA TUXEDO system, not both. See the TxRPC Guide for a discussion of integrating BEA TUXEDO system/TxRPC stubs and OSF/DCE stubs in a single program.

When linking a program using this interface, `$TUXDIR/lib/libtrpc.a` must be included.

Examples

Here 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 tidl(1), abort(2), userlog(3), *TUXEDO TxRPC Guide*

tuxgetenv(3)

Name: `tuxgetenv`-return 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 system is supported, including those platforms that don't normally have environment variables.

Note that `tuxgetenv` is case-sensitive.

Return Values: `tuxgetenv`() returns a pointer to the string if present and a null pointer otherwise.

Portability: On MS Windows, this function overcomes the inability to share environment variables between an application and a Dynamic Link Library. The TUXEDO /WS DLL maintains an environment copy for each application that is attached to it. This associated environment and context information is destroyed when `tpterm`(3c) is called from a Windows application. The value of an environment variable could be changed after the application program calls `tpterm`(3c).

It is recommended that upper case variable names be used for the DOS, Windows, OS/2, and NetWare environments. (`tuxreadenv`(3c) converts all environment variable names to upper case.)

See Also: `tuxputenv`(3), `tuxreadenv`(3)

tuxputenv(3)

Name tuxputenv(3)-change or add value to 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 system is supported, including those platforms that don't normally have environment variables.

Note that tuxputenv is case-sensitive.

Return Values tuxputenv() returns a non-zero integer if it was unable to obtain enough space via malloc for an expanded environment, otherwise 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 system /WS DLL maintains an environment copy for each application that is attached to it. This associated environment and context information is destroyed when tpterm(3c) is called from a Windows application. The value of an environment variable could be changed after the application program calls tpterm(3c).

We recommend using upper case variable names for the DOS, Windows, and OS/2, environments. (tuxreadenv(3c) converts all environment variable names to upper case.)

See Also tuxgetenv(3), tuxreadenv(3)

tuxreadenv(3)

Name — `tuxreadenv`-add 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`(3) and can be reset using `tuxputenv`(3).

The format of the environment file is as follows.

- ♦ Any leading space or tab characters on each line are ignored and are 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 file name is used. The fixed file names 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 file name does not exist, or if a non-null label does not exist.

Example Here 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 `tuxreadenv`() returns non-zero if it was unable to obtain enough space via `malloc` for an expanded environment or was unable to open and read a non-NULL filename, otherwise zero.

Portability In the DOS, Windows, OS/2, and NetWare environments, `tuxreadenv`() converts all environment variable names to upper case.

See Also `tuxgetenv`(3), `tuxputenv`(3)

tx_begin(3)

Name
`tx_begin`-begin 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`(3)) 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`(3) or `tx_rollback`(3) 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.

Optional Set-up
`tx_set_transaction_timeout`(3)

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`(3), `tx_open`(3), `tx_rollback`(3), `tx_set_transaction_timeout`(3)

Warnings
XA-compliant resource managers must be successfully opened to be included in the global transaction. (See `tx_open`(3) for details.) Both the X/Open TX interface and the X-Windows system defines the type XID. It is not possible to use both X-Windows calls and TX calls in the same file.

tx_close(3)

Name `tx_close`-close 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.

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`(3)

Warnings Both the X/Open TX interface and the X-Windows system defines the type XID. It is not possible to use both X-Windows calls and TX calls in the same file.

tx_commit(3)

Name
`tx_commit`-commit 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`(3)) 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).

OPTIONAL SET-UP

- `tx_set_commit_return`(3)
- `tx_set_transaction_control`(3)
- `tx_set_transaction_timeout`(3)

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(3), tx_set_commit_return(3), tx_set_transaction_control(3), tx_set_transaction_timeout(3)

Warnings: Both the X/Open TX interface and the X-Windows system defines the type XID. It is not possible to use both X-Windows calls and TX calls in the same file.

tx_info(3)

Name `tx_info`-return 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 <tx.h> 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`(3) after the current transaction was begun may have changed its value.

If *`info`* is null, no TXINFO structure is returned.

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`(3)).

`[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`(3), `tx_set_commit_return`(3), `tx_set_transaction_control`(3), `tx_set_transaction_timeout`(3)

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 defines the type XID. It is not possible to use both X-Windows calls and TX calls in the same file.

tx_open(3)

Name tx_open-open 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(3)) are allowed. However, such subsequent calls will return success, and the TM will not attempt to re-open any RMs.

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(3) 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`(3)

Warnings Both the X/Open TX interface and the X-Windows system defines the type XID. It is not possible to use both X-Windows calls and TX calls in the same file.

tx_rollback(3)

Name
: `tx_rollback`-roll 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`(3)) 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).

OPTIONAL SET-UP

- `tx_set_transaction_control`(3)
- `tx_set_transaction_timeout`(3)

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(3), tx_set_transaction_control(3), tx_set_transaction_timeout(3)

Warnings
: Both the X/Open TX interface and the X-Windows system defines the type XID. It is not possible to use both X-Windows calls and TX calls in the same file.

tx_set_commit_return(3)

Name `tx_set_commit_return`-set *`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`(3) 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`.

Following are the valid settings for *`when_return`*.

`TX_COMMIT_DECISION_LOGGED`
: This flag indicates that `tx_commit`(3) 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`(3). 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`(3). 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`(3) should return after the two-phase commit protocol has finished completely. This setting allows the caller of `tx_commit`(3) to see return codes that indicate that a transaction had or may have had heuristic results.

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`(3)).

`[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`(3), `tx_open`(3), `tx_info`(3)

Warnings

Both the X/Open TX interface and the X-Windows system defines the type XID. It is not possible to use both X-Windows calls and TX calls in the same file.

tx_set_transaction_control(3)

Name tx_set_transaction_control-set *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(3) and tx_rollback(3) 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.

Following are the valid settings for *control*.

TX_UNCHAINED

This flag indicates that tx_commit(3) and tx_rollback(3) should not start a new transaction before returning to their caller. The caller must issue tx_begin(3) to start a new transaction.

TX_CHAINED

This flag indicates that tx_commit(3) and tx_rollback(3) should start a new transaction before returning to their caller.

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(3)).

[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`(3), `tx_commit`(3), `tx_open`(3), `tx_rollback`(3), `tx_info`(3)

Warnings
Both the X/Open TX interface and the X-Windows system defines the type XID. It is not possible to use both X-Windows calls and TX calls in the same file.

tx_set_transaction_timeout(3)

Name
`tx_set_transaction_timeout`-set *`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`(3) and `tx_commit`(3) or `tx_rollback`(3). `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.

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 in an improper context. For example, the caller has not yet called `tx_open`(3).

[`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`(3), `tx_commit`(3), `tx_open`(3), `tx_rollback`(3), `tx_info`(3)

Warnings

Both the X/Open TX interface and the X-Windows system defines the type XID. It is not possible to use both X-Windows calls and TX calls in the same file.

userlog(3)

Name: `userlog`-write a message to the BEA TUXEDO 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 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 userlog, it first writes an additional message indicating the associated BEA TUXEDO 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 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 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 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 system to record a variety of events.

The userlog mechanism is entirely independent of any database transaction logging mechanism.

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 reference manual

Usignal(3)

Name

Usignal-signal handling in a BEA TUXEDO 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 system software require concurrent access to data structures in shared memory. Processes accessing the shared data structures run in user mode, and are thus interruptable 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 system code. Additionally, they are available to applications to prevent the untimely arrival of a signal.

The idea behind the BEA TUXEDO 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 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.

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(2)` 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 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 system clients when they call `tpinit`() and for BEA TUXEDO 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 (i.e. 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`(2) 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 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`(2).

`Usignal` is not available under DOS operating systems.

Files Usignal.h

See Also `signal`(2) in a UNIX System reference manual

Uunix_err(3)

Name `Uunix_err`-print UNIX system call error

Synopsis
```
#include Uunix.h

void Uunix_err(s)
char *s;
```

Description When a BEA TUXEDO 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 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).

Examples
```
#include Uunix.h
extern int Uunixerr, errno;

  ..........
   if((fd=open("myfile", 3, 0660)) == -1)
       {
       Uunixerr = UOPEN;
       Uunix_err("myprog");
       exit(1);
       }
```

xdr(3I)

Name

`xdr`-library routines for external data representation

Description

`XDR` routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Data for communications calls are transmitted using these routines.

Index to Routines

The following table lists `XDR` routines and the manual reference pages on which they are described:

XDR Routines

XDR Routine	Manual Reference Page
`xdr_array`	`xdr_complex(3I)`
`xdr_bool`	`xdr_simple(3I)`
`xdr_bytes`	`xdr_complex(3I)`
`xdr_char`	`xdr_simple(3I)`
`xdr_destroy`	`xdr_create(3I)`
`xdr_double`	`xdr_simple(3I)`
`xdr_enum`	`xdr_simple(3I)`
`xdr_float`	`xdr_simple(3I)`
`xdr_free`	`xdr_simple(3I)`
`xdr_getpos`	`xdr_admin(3I)`
`xdr_inline`	`xdr_admin(3I)`
`xdr_int`	`xdr_simple(3I)`
`xdr_long`	`xdr_simple(3I)`
`xdr_opaque`	`xdr_complex(3I)`
`xdr_pointer`	`xdr_complex(3I)`
`xdr_reference`	`xdr_complex(3I)`
`xdr_setpos`	`xdr_admin(3I)`
`xdr_short`	`xdr_simple(3I)`
`xdr_string`	`xdr_complex(3I)`

XDR Routines

XDR Routine	Manual Reference Page
`xdr_u_char`	`xdr_simple(3I)`
`xdr_u_long`	`xdr_simple(3I)`
`xdr_u_short`	`xdr_simple(3I)`
`xdr_union`	`xdr_complex(3I)`
`xdr_vector`	`xdr_complex(3I)`
`xdr_void`	`xdr_simple(3I)`
`xdr_wrapstring`	`xdr_complex(3I)`
`xdrmem_create`	`xdr_create(3I)`
`xdrstdio_create`	`xdr_create(3I)`

See Also `xdr_admin`(3I), `xdr_complex`(3I), `xdr_create`(3I), `xdr_simple`(3I)

xdr_admin(3I)

Name `xdr_admin`, `xdr_getpos`, `xdr_inline`, `xdr_setpos`-library routines for external data representation

Description `XDR` library routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Protocols such as communications calls use these routines to describe the format of the data.

These routines deal specifically with the management of the `XDR` stream.

Routines `#include <rpc/xdr.h>`

`u_int xdr_getpos(const XDR *xdrs)`
A macro that invokes the get-position routine associated with the `XDR` stream, *`xdrs`*. The routine returns an unsigned integer, which indicates the position of the `XDR` byte stream. A desirable feature of `XDR` streams is that simple arithmetic works with this number, although the `XDR` stream instances need not guarantee this. Therefore, applications written for portability should not depend on this feature.

`long * xdr_inline(XDR *xdrs, const int len)`
A macro that invokes the in-line routine associated with the `XDR` stream, *`xdrs`*. The routine returns a pointer to a contiguous piece of the stream's buffer; *`len`* is the byte length of the desired buffer. Note: pointer is cast to `long *`. Warning: `xdr_inline` may return `NULL` (`0`) if it cannot allocate a contiguous piece of a buffer. Therefore the behavior may vary among stream instances; it exists for the sake of efficiency, and applications written for portability should not depend on this feature.

`bool_t xdr_setpos(XDR *xdrs, const u_int pos)`
A macro that invokes the set position routine associated with the `XDR` stream *`xdrs`*. The parameter *`pos`* is a position value obtained from `xdr_getpos`. This routine returns `1` if the `XDR` stream was repositioned, and `0` otherwise. Warning: it is difficult to reposition some types of `XDR` streams, so this routine may fail with one type of stream and succeed with another. Therefore, applications written for portability should not depend on this feature.

See Also `xdr_complex`(3I), `xdr_create`(3I), `xdr_simple`(3I).

xdr_complex(3I)

Name

xdr_complex: xdr_array, xdr_bytes, xdr_opaque, xdr_pointer, xdr_reference, xdr_string, xdr_union, xdr_vector, xdr_wrapstring-library routines for external data representation

Description

XDR library routines allow C programmers to describe complex data structures in a machine-independent fashion. Protocols such as communications calls use these routines to describe the format of the data. These routines are the XDR library routines for complex data structures. They require the creation of XDR stream [see xdr_create(3I)].

Routines

```
#include <rpc/xdr.h>
```

```
bool_t xdr_array(XDR *xdrs, caddr_t *arrp, u_int *sizep, const
u_int maxsize, const u_int elsize, const xdrproc_t elproc)
```

xdr_array translates between variable-length arrays and their corresponding external representations. The parameter *arrp* is the address of the pointer to the array, while *sizep* is the address of the element count of the array; this element count cannot exceed *maxsize*. The parameter *elsize* is the sizeof each of the array's elements, and *elproc* is an XDR routine that translates between the array elements' C form and their external representation. This routine returns 1 if it succeeds, 0 otherwise.

```
bool_t xdr_bytes(XDR *xdrs, char **sp, u_int *sizep, const
u_int maxsize)
```

xdr_bytes translates between counted byte strings and their external representations. The parameter *sp* is the address of the string pointer. The length of the string is located at address *sizep*; strings cannot be longer than *maxsize*. This routine returns 1 if it succeeds, 0 otherwise.

```
bool_t xdr_opaque(XDR *xdrs, caddr_t cp, const u_int cnt)
```

xdr_opaque translates between fixed size opaque data and its external representation. The parameter *cp* is the address of the opaque object, and *cnt* is its size in bytes. This routine returns 1 if it succeeds, 0 otherwise.

```
bool_t xdr_pointer(XDR *xdrs, char **objpp, u_int objsize,
const xdrproc_t xdrobj)
```

Like xdr_reference except that it serializes NULL pointers, whereas xdr_reference does not. Thus, xdr_pointer can represent recursive data structures, such as binary trees or linked lists.

`bool_t xdr_reference(XDR *xdrs, caddr_t *pp, u_int size,`
`const xdrproc_t proc)`

: `xdr_reference` provides pointer chasing within structures. The parameter *`pp`* is the address of the pointer; *`size`* is the `sizeof` the structure that *`*pp`* points to; and *`proc`* is an `XDR` procedure that translates the structure between its C form and its external representation. This routine returns `1` if it succeeds, `0` otherwise. Warning: this routine does not understand `NULL` pointers. Use `xdr_pointer` instead.

`bool_t xdr_string(XDR *xdrs, char **sp, const u_int maxsize)`

: `xdr_string` translates between C strings and their corresponding external representations. Strings cannot be longer than *`maxsize`*. Note: *`sp`* is the address of the string's pointer. This routine returns `1` if it succeeds, `0` otherwise.

`bool_t xdr_union(XDR *xdrs, enum_t *dscmp, char *unp,`
`const struct xdr_discrim *choices, const bool_t (*defaultarm)(const`
`XDR *, const char *, const int))`

: `xdr_union` translates between a discriminated C `union` and its corresponding external representation. It first translates the discriminant of the union located at *`dscmp`*. This discriminant is always an `enum_t`. Next the union located at *`unp`* is translated. The parameter *`choices`* is a pointer to an array of `xdr_discrim` structures. Each structure contains an ordered pair of [*`value, proc`*]. If the union's discriminant is equal to the associated *`value`*, then the *`proc`* is called to translate the union. The end of the `xdr_discrim` structure array is denoted by a routine of value `NULL`. If the discriminant is not found in the *`choices`* array, then the *`defaultarm`* procedure is called (if it is not `NULL`). Returns `1` if it succeeds, `0` otherwise.

`bool_t xdr_vector(XDR *xdrs, char *arrp, const u_int size,`
`const u_int elsize, const xdrproc_t elproc)`

: `xdr_vector` translates between fixed-length arrays and their corresponding external representations. The parameter *`arrp`* is the address of the pointer to the array, while *`size`* is is the element count of the array. The parameter *`elsize`* is the `sizeof` each of the array's elements, and *`elproc`* is an `XDR` routine that translates between the array elements' C form and their external representation. This routine returns `1` if it succeeds, `0` otherwise.

`bool_t xdr_wrapstring(XDR *xdrs, char **sp)`

: A routine that calls xdr_string(*`xdrs`*, *`sp`*, *`maxuint`*); where *`maxuint`* is the maximum value of an unsigned integer. Many routines, such as `xdr_array`, `xdr_pointer` and `xdr_vector` take a function pointer of type `xdrproc_t`, which takes two arguments. `xdr_string`, one of the most frequently used routines, requires three arguments, while `xdr_wrapstring` only requires two. For these routines, `xdr_wrapstring` is desirable. This routine returns `1` if it succeeds, `0` otherwise.

See Also `xdr_admin`(3I), `xdr_create`(3I), `xdr_simple`(3I).

xdr_create(3I)

Name

`xdr_create`: `xdr_destroy`, `xdrmem_create`, `xdrstdio_create`-library routines for external data representation stream creation

Description

`XDR` library routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Protocols such as communications calls use these routines to describe the format of the data.

These routines deal with the creation of `XDR` streams. `XDR` streams have to be created before any data can be translated into `XDR` format.

Routines

```
#include <rpc/xdr.h>
```

```
void xdr_destroy(XDR *xdrs)
```

A macro that invokes the destroy routine associated with the `XDR` stream, *`xdrs`*. Destruction usually involves freeing private data structures associated with the stream. Using *`xdrs`* after invoking `xdr_destroy` is undefined.

```
void xdrmem_create(XDR *xdrs, const caddr_t addr, const u_int size,
 const enum xdr_op op)
```

This routine initializes the `XDR` stream object pointed to by *`xdrs`*. The stream's data is written to, or read from, a chunk of memory at location *`addr`* whose length is no more than *`size`* bytes long. The *`op`* determines the direction of the `XDR` stream (either `XDR_ENCODE`, `XDR_DECODE`, or `XDR_FREE`).

```
void xdrstdio_create(XDR *xdrs, FILE *file, const enum xdr_op op)
```

This routine initializes the `XDR` stream object pointed to by *`xdrs`*. The `XDR` stream data is written to, or read from, the standard I/O stream *`file`*. The parameter *`op`* determines the direction of the `XDR` stream (either `XDR_ENCODE`, `XDR_DECODE`, or `XDR_FREE`). Warning: the destroy routine associated with such `XDR` streams calls `fflush` on the *`file`* stream, but never `fclose` [see `fclose`(3S)].

See Also

`fclose`(3S), `read`(2), `rpc`(3I), `write`(2), `xdr_admin`(3I), `xdr_complex`(3I), `xdr_simple`(3I).

xdr_simple(3I)

Name
`xdr_simple`: `xdr_bool`, `xdr_char`, `xdr_double`, `xdr_enum`, `xdr_float`, `xdr_free`, `xdr_int`, `xdr_long`, `xdr_short`, `xdr_u_char`, `xdr_u_long`, `xdr_u_short`, `xdr_void`-library routines for external data representation

Description
`XDR` library routines allow C programmers to describe simple data structures in a machine-independent fashion. Protocols such as communications calls use these routines to describe the format of the data.

These routines require the creation of `XDR` streams [see xdr_create(3I)].

Routines
```
#include <rpc/xdr.h>
```

```
bool_t xdr_bool(XDR *xdrs, bool_t *bp)
```
`xdr_bool` translates between booleans (C integers) and their external representations. When encoding data, this filter produces values of either `1` or `0`. This routine returns `1` if it succeeds, `0` otherwise.

```
bool_t xdr_char(XDR *xdrs, char *cp)
```
`xdr_char` translates between C characters and their external representations. This routine returns `1` if it succeeds, `0` otherwise. Note: encoded characters are not packed, and occupy 4 bytes each. For arrays of characters, it is worthwhile to consider `xdr_bytes`, `xdr_opaque` or `xdr_string` [see `xdr_bytes`, `xdr_opaque` and `xdr_string` in `xdr_complex`(3I)].

```
bool_t xdr_double(XDR *xdrs, double *dp)
```
`xdr_double` translates between C `double` precision numbers and their external representations. This routine returns `1` if it succeeds, `0` otherwise.

```
bool_t xdr_enum(XDR *xdrs, enum_t *ep)
```
`xdr_enum` translates between C `enum`s (actually integers) and their external representations. This routine returns `1` if it succeeds, `0` otherwise.

```
bool_t xdr_float(XDR *xdrs, float *fp)
```
`xdr_float` translates between C `float`s and their external representations. This routine returns `1` if it succeeds, `0` otherwise.

```
void xdr_free(xdrproc_t proc, char *objp)
```
Generic freeing routine. The first argument is the `XDR` routine for the object being freed. The second argument is a pointer to the object itself. Note: the pointer passed to this routine is not freed, but what it points to is freed (recursively).

`bool_t xdr_int(XDR *xdrs, int *ip)`
: `xdr_int` translates between C integers and their external representations. This routine returns `1` if it succeeds, `0` otherwise.

`bool_t xdr_long(XDR *xdrs, long *lp)`
: `xdr_long` translates between C `long` integers and their external representations. This routine returns `1` if it succeeds, `0` otherwise.

`bool_t xdr_short(XDR *xdrs, short *sp)`
: `xdr_short` translates between C `short` integers and their external representations. This routine returns `1` if it succeeds, `0` otherwise.

`bool_t xdr_u_char(XDR *xdrs, char *ucp)`
: `xdr_u_char` translates between `unsigned` C characters and their external representations. This routine returns `1` if it succeeds, `0` otherwise.

`bool_t xdr_u_long(XDR *xdrs, unsigned long *ulp)`
: `xdr_u_long` translates between C `unsigned long` integers and their external representations. This routine returns `1` if it succeeds, `0` otherwise.

`bool_t xdr_u_short(XDR *xdrs, unsigned short *usp)`
: `xdr_u_short` translates between C `unsigned short` integers and their external representations. This routine returns `1` if it succeeds, `0` otherwise.

`bool_t xdr_void(void)`
: This routine always returns `1`. It may be passed to RPC routines that require a function parameter, where nothing is to be done.

See Also `rpc`(3I), `xdr_admin`(3I), `xdr_complex`(3I), `xdr_create`(3I).