BEA eLink Adapter Development Kit

User Guide

BEA eLink Adapter Development Kit Version 1.1
Document Edition 1.1
April 2000

BEA eLink Adapter Development Kit

Document Edition	Part Number	Date	Software Version
1.1	Not Applicable	April 2000	BEA eLink Adapter Development Kit Version 1.1
1.0	Not Applicable	January 2000	BEA eLink Adapter Development Kit Version 1.0

Contents

About This Document

This document explains what the eLink Adapter Development Kit is and describes how to use it for designing adapters to third-party enterprise applications such as ERP, CRM, and Supply Chain Management.

This document covers the following topics:

- Chapter 1, "Understanding The BEA eLink System," a brief description of the BEA eLink system.
- Chapter 2, "Understanding the BEA eLink Platform," a brief description of the BEA eLink platform.
- Chapter 3, "Understanding Adapter Architecture and Design," a brief description of adapter architecture and design.
- Chapter 4, "Installing the eLink ADK and Sample Adapters," a description of what is included in the eLink Adapter Development Kit and how to install it.
- Chapter 5, "Configuring and Running the Sample Adapters," a description of how to configure and run the sample adapters.
- Appendix A, "eLink Adapter Development Kit References," a description of eLink Adapter configuration processing API, hash table API, utility functions and macros, and definitions and typedefs.
- Appendix B, "ATMI References," a description of commonly used Tuxedo ATMI functions for the eLink Adapter Development Kit.
- Appendix C, "FML32 API," a description of commonly used Tuxedo FML32 API functions for the eLink Adapter Development Kit.
- Appendix D, "Tuxedo Commands," a description of some of the commonly used Tuxedo commands for the eLink Adapter Development Kit.
- Appendix E, "Servopts," a description of the Tuxedo servopts function.

- Appendix F, "Error Messages," a description of eLink Adapter Development Kit error messages and recommended actions.
- Glossary

What You Need to Know

This document is intended mainly for Application Programmers who will configure and set up the BEA eLink Adapter Development Kit and eLink services to create software components called adapters, which are used to integrate third-party applications to the eLink system. It is assumed that the programmer has experience with the C language. Experience with BEA Tuxedo is an asset, but not necessary.

e-docs Web Site

BEA product documentation is available in both PDF and HTML format on the BEA corporate Web site. From the BEA Home page, click on Product Documentation or go directly to the "e-docs" Product Documentation page at http://www.e-docs.beasys.com.

How to Print the Document

You can print a copy of the HTML document, one file at a time, from your Web browser by selecting File|Print, or you can print the PDF document. Open the PDF file in the Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the eLink Adapter Development Kit 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/. Refer to the eLink Adapter Development Kit *Release Notes* for more detailed information about viewing and printing the documentation.

Related Information

The following BEA Tuxedo documents contain information that is relevant to using the eLink Adapter Development Kit.

- *BEA Tuxedo Administering the BEA Tuxedo System*
- *BEA Tuxedo Application Development Guide*
- *BEA Tuxedo FML Programmer's Guide*
- *BEA Tuxedo Programmer's Guide*
- *BEA Tuxedo Reference Manual*

For more information about Tuxedo, refer to the BEA Tuxedo Online Documentation CD at http://edocs.beasys.com/tuxedo/tux65/index.htm.

Contact Us!

Your feedback on the eLink Adapter Development Kit 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 eLink Adapter Development Kit documentation. In your e-mail message, please indicate that you are using the documentation for the BEA eLink Adapter Development Kit Product Version: 1.1 release.

If you have any questions about this version of BEA eLink Adapter Development Kit, or if you have problems installing and running BEA eLink Adapter Development Kit, 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 typographic conventions are used throughout this document.

Convention	Item
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
italics	Indicates emphasis or book titles.
blue text	Indicates a hyperlink to a cross-reference.

Convention	Item

`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`*`]...`

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

Convention	Item
`...`	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.

1 Understanding The BEA eLink System

This section discusses the following topics:

- BEA eLink Solution Overview
- BEA eLink Adapter Overview
- BEA eLink Adapter Development Kit Overview
- BEA eLink Platform Architecture

BEA eLink Solution Overview

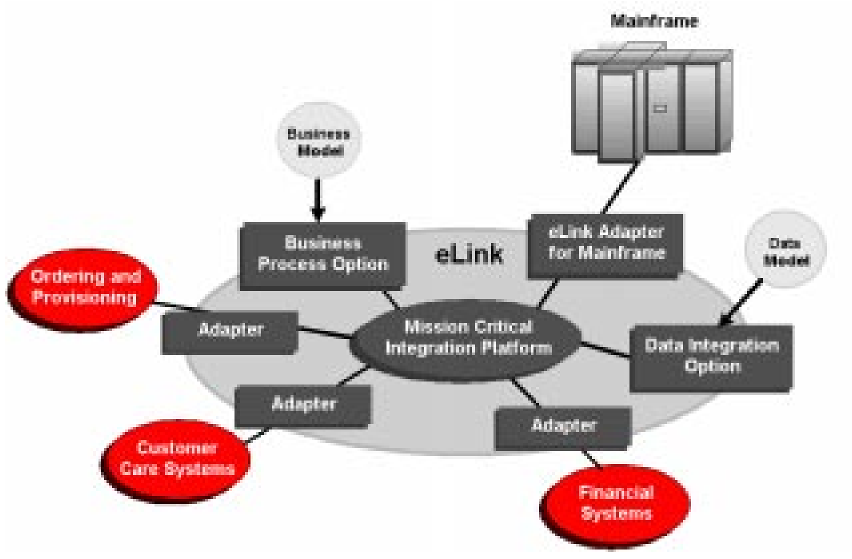
BEA eLink™ provides an open Enterprise Application Integration (EAI) solution that allows applications throughout organizations to communicate seamlessly. Using EAI, you gain the long-term flexibility and investment protection you need to keep up with today's ever-changing business environment.

Typically, companies use packaged applications to automate internal operations, such as financial, manufacturing, or human resources. While they successfully address the needs of these specific areas, these proprietary platforms often do not work together. To compete today, you need a much greater exchange of information. Systems need to communicate at a process level within your own organization, as well as with customer's and supplier's systems. BEA eLink Platform is the underlying basis of

BEA eLink, a family of off-the-shelf enterprise application integration (EAI) products that leverage the BEA transaction platform to integrate existing legacy applications with customer-focused and business-to-business e-commerce initiatives.

BEA eLink Platform provides a proven infrastructure for integrating applications within the enterprise and across the Web. BEA eLink Platform ensures high-performance, secure transactions and transparent access to mission-critical applications and information throughout the enterprise and across the Web. Figure 1-1 illustrates the eLink logical architecture and shows where the eLink Adapters fit into the process.

Figure 1-1 BEA eLink Solution Illustration



The entire BEA eLink family (including all options and adapters) is highly scalable. Multiple instances of BEA eLink components can collaborate so that work is divided between eLink domains. BEA eLink includes Simple Network Management Protocol (SNMP) integration for enterprise management.

The current BEA eLink Platform leverages the BEA Tuxedo infrastructure because it is based on a service-oriented architecture. Both BEA Tuxedo and BEA eLink communicate directly with each other and with other applications through the use of services. Multiple services are grouped into "application servers" or "servers". The terms Tuxedo services/servers and eLink services/servers can be used interchangeably. Because this document is specifically addressing the eLink family, the terms “eLink service” and “eLink server” are used throughout.

The BEA eLink Platform complies with the Open Group’s X/Open standards including support of the XA standard for two-phase commit processing, the X/Open **ATMI** API, and XPG standards for language internationalization. C, C++, COBOL, and Java are supported. The BEA eLink Platform connects to any RDBMS, OODBMS, file manager or queue manager, including a supplied XA-compliant queueing subsystem.

The following components operate with BEA eLink Platform:

- ♦ The **Data Integration Option** translates data models used by different applications into a common data format. It provides a cost-effective alternative to writing or generating programs to perform this function. It also handles complex translation with great power and scalability. The DIO leverages technology based on the TSI Mercator product, which is integrated with eLink.
- ♦ The **Business Process Option** helps automate tasks in the distributed global business process and dynamically responds to business events and exceptions. The BPO is currently implemented by integrating eLink with technology based on InConcert workflow management software.
- ■ An **eLink Adapter** provides the interface between the BEA eLink Platform and external applications with out-of-the-box functionality.

BEA eLink Adapter Overview

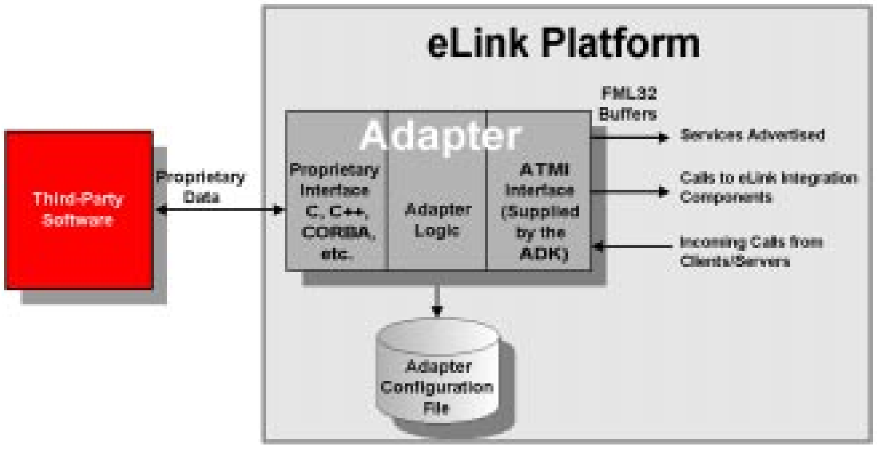
eLink Adapters provide a communication path between the eLink Platform and third-party applications such as PeopleSoft or SAP. eLink Adapters are implemented as eLink Platform servers. **Servers** are software modules responsible for processing service requests made by requestors and potentially sending replies back to the originator of the request. **Services** are provided by code that accesses the **business logic** of a third-party application.

An eLink Adapter must accomplish at least two things. First, it must normalize the communication between the two components. Second, it should advertise business level services supported by the third-party application into the eLink Platform environment. In addition, application to eLink Adapters must publish events with associated data on behalf of the third-party application in the eLink Platform environment. If the third-party application allows scripting or other extensions, it should be capable of invoking known eLink Platform services advertised by other third-party applications.

An adapter may support both the eLink bound paths (application to eLink) and third-party application bound paths (eLink to application); however, it is recommended that an adapter be implemented for only one path. If both application to eLink and eLink to application paths are required, they should be implemented as two separate servers. Separate servers allow for a more flexible solution because the usage requirements of eLink-bound and application-bound adapters may vary widely.

eLink Adapters use **FML32**, a BEA native data structure, to communicate with other components in the integration environment. The third-party native interface is used to communicate with the third-party application. eLink Adapters act as a “translator” between the third-party API and the BEA **ATMI**. This translation is facilitated by use of the ADK API.

Figure 1-2 eLink Adapter Illustration


eLink Platform
Third-Party Software
Proprietary Data
Adapter
Proprietary Interface C, C++, CORBA, etc.
Adapter Logic
ATMI Interface (Supplied by the ADK)
FML32 Buffers
Services Advertised
Calls to eLink Integration Components
Incoming Calls from Clients/Servers
Adapter Configuration File

BEA eLink Adapter Development Kit Overview

The eLink Adapter Development Kit (ADK) is a set of tools and libraries that allow BEA, our partners, and our customers to build adapters that interact with the eLink Platform. The ADK helps programmers build adapters for third-party software without in-depth knowledge of the eLink Platform.

The ADK consists of:

- **A sample Application to eLink Adapter.** The sample Application to eLink Adapter contains code to generate the sample application to eLink server, a demo server needed for testing, sample `UBBCONFIG` file (eLink configuration), `.MAK` files for all supported UNIX platforms, as well as `.BAT` and `.SCR` files to setup the environment.
- **A sample eLink to Application Adapter.** The sample eLink to Application Adapter contains code to generate demo servers, a demo client to be used for testing, sample `UBBCONFIG` file (eLink configuration) and `.CFG` files (adapter specific configuration), `.MAK` files for all supported UNIX platforms, as well as `.BAT` and `.SCR` files to setup the environment and automate the build process.
- **A sample E-Mail Adapter.** The sample E-Mail Adapter contains code to generate the sample E-Mail server and client, which can be used for sending E-Mail. A template is available to help the user develop functionality for receiving E-Mail.
- **Required include files (.H); shared libraries, (.SL) for HP-UX 10.20 and 11.0, (.S0) for Solaris 2.6, (.a) for AIX 4.3.x; and a dynamic link library (.DLL) and import library (.LIB) for Windows NT.** These libraries support the sample eLink to application and application to eLink Adapters, as well as adapter development. The .LIB file is compiled with MS VC++ v5.x.
- **This documentation.** The documentation includes general information about the eLink Platform, design information for adapters, and instructions for running the sample adapter.

BEA eLink Platform Architecture

The eLink Platform communications application programming interface, Application to Transaction Manager Interface (**ATMI**), is a collection of runtime services that can be called directly by a C (or COBOL) application. These runtime services provide support for communications, distributed transactions, and system management.

The Management Information Base (**MIB**) maintains a virtual repository of all the configuration and operational information for a runtime eLink environment. The eLink services are implemented using a shared bulletin board (BB) that contains configuration information. This is the dynamic part of the eLink. **Servers** advertise their **services** in the Bulletin Board. The Bulletin Board Liaison (BBL) is an administrative eLink server that is the keeper of the Bulletin Board. There is a BBL on every machine participating in the integration infrastructure; the BBL coordinates changes to the local copy of the MIB. The Distinguished Bulletin Board Liaison (DBBL) is responsible for propagating global changes to the MIB and is the keeper of the static part of the MIB. The MASTER node is the computer where the DBBL runs.

Administrators use an ASCII file to specify eLink system configuration. This file, called the **`UBBCONFIG`** file, is used as input by the configuration loading utility, `tmloadcf`. The `tmloadcf` utility generates a binary version of the configuration called the `tuxconfig` file. This binary file is used by the system to construct the Bulletin Board and contains the persistent part of the MIB.

Servers are processes that provide one or more services. They continually check their message queue for service requests and dispatch them to the appropriate third-party application. A service is the name of a server interface. Many servers can support a single service, thereby providing for load balancing and a fail-safe mechanism. The mapping of services to servers is recorded in the Bulletin Board. When a service request is made, the Bulletin Board forwards the request to a server (eLink Adapter) that advertises that service. An eLink server advertises a service by posting its name in the Bulletin Board.

2 Understanding the BEA eLink Platform

This chapter discusses the following topics:

- ATMI Runtime Services
- FML32
- eLink Commands
- Hardware Requirements for the eLink Platform
- Special Instructions for Installing the Tuxedo Core
- Preparing the License File
- About the Tuxedo Simple Application

Before you install the BEA eLink Platform, which is available separately, it is helpful to understand the Application to Transaction Manager Interface (**ATMI**), the 32-bit Field Manipulation Language (**FML32**), FML buffers, and some commonly used eLink commands. The ATMI runtime services provide support for communications, distributed transactions, and system management. FML32 is a BEA native data structure and function library that allows associative access to fields of a data record. Commands are used to configure and administer the BEA eLink environment.

ATMI Runtime Services

The eLink Platform ATMI is a collection of runtime services that can be called directly by a C (or COBOL) application. The ATMI is a compact set of primitives used to open and close resources, begin and end transactions, allocate and free buffers, and provide the communication between adapters and other requestors or responders.

Following is a list of ATMI primitives for the C binding. For more complete details of the ATMI primitives you will most commonly use, see Appendix B, "ATMI References." See the *BEA Tuxedo Reference Guide* at http://edocs.beasys.com/tuxedo/tux65/index.htm for detailed information on all the ATMI primitives.

Table 2-1 ATMI Primitives for the C Binding

API Group	C API Name	Detailed in Appendix B	Description
Client Membership	tpchkauth		Check if authentication is needed
	tpinit	*	Used by a client to join an application
	tpterm	*	Used by a client to leave an application
Buffer Management	tpalloc	*	Create a message
	tprealloc	*	Resize a message
	tpfree	*	Free a message
	tptypes	*	Get a message type and subtype
Message Priority	tpgprio		Get the priority of the last request
	tpsprio		Set priority of the next request
Request/Response	tpcall	*	Synchronous request/response to **service**
	tpacall	*	Asynchronous request
	tpgetreply	*	Receive asynchronous response
	tpcancel	*	Cancel asynchronous request
Conversational	tpconnect		Begin a conversation with a **service**
	tpdiscon		Abnormally terminate a conversation
	tpsend		Send a message in a conversation
	tprecv		Receive a message in a conversation

Table 2-1 ATMI Primitives for the C Binding

API Group	C API Name	Detailed in Appendix B	Description
Reliable Queueing	tpenqueue		Enqueue a message to an application queue
	tpdequeue		Dequeue a message to an application queue
Event-based	tpnotify		Send unsolicited message to a client
	tpbroadcast		Send message to several clients
	tpsetunsol		Set unsolicited message callback
	tpchkunsol		Check arrival of unsolicited message
	tppost		Post an event message
	tpsubscribe		Subscribe to event messages
	tpunsubscribe		Unsubscribe to event messages
Transaction Management	tpbegin		Begin a transaction
	tpcommit		Commit the current transaction
	tpabort		Rollback the current transaction
	tpgetlev		Check if in transaction mode
	tpsuspend		Suspend the current transaction
	tpresume		Resume a transaction
	tpscmt		Control commit return
Service Entry and Return	tpsvrinit	*	**Server** initialization
	tpsvrdone	*	**Server** termination
	tpreturn	*	End **service** function
	tpforward	*	Forward request
Dynamic Advertisement	tpadvertise	*	Advertise a **service** name
	tpunadvertise	*	Unadvertise a **service** name
Resource Management	tpopen		Open a resource manager
	tpclose		Close a resource manager

FML32

FML is a set of C language functions for defining and manipulating storage structures called fielded buffers, which contain attribute-value pairs called fields. The attribute is the field's identifier and the associated value represents the field's data content.

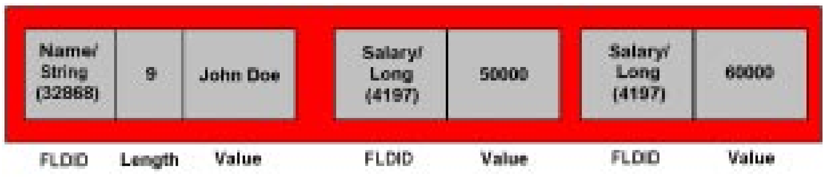
FML32 uses 32-bit values for the field lengths and identifiers. BEA eLink Adapters use FML32. FML32 allows for about 30 million fields, and field and buffer lengths of up to about 2 billion bytes. The definitions, types, and function prototypes for FML32 are located in `fml32.h` and functions are located in `-lfml32`. All definitions, types, and function names for FML32 have a "32" suffix (for example, `MAXFBLEN32`, `FLDID32`, `Fchg32`). Also the environment variables are suffixed with "32" (for example, `FLDTBLDIR32`, `FIELDTBLS32`).

Note: FML has two interfaces. The original FML interface is based on 16-bit values for the length of fields and for containing information identifying fields. The original interface should not be used when creating eLink Adapters.

FML Buffers

A fielded buffer is composed of field identifier and field value pairs for fixed length fields (for example, long, short), and field identifier, field length, and field value triples for varying length fields.

Figure 2-1 Example of a Fielded Buffer



A field identifier is a tag for an individual data item in a fielded buffer. The field identifier consists of the name of the field number and the type of data in the field. The field number must be in the range 1 to 33,554,431 inclusive for FML32, and the type definition for a field identifier is `FLDID32`.

Field numbers 1 to 100 are reserved for system use and should be avoided. The field types can be any of the standard C language types: `short`, `long`, `float`, `double`, and `char`. Two other types are also supported: `string` (a series of characters ending with a null character) and `carray` (character arrays). These types are defined in `fml32.h` as `FLD_SHORT`, `FLD_LONG`, `FLD_CHAR`, `FLD_FLOAT`, `FLD_DOUBLE`, `FLD_STRING`, and `FLD_CARRAY`.

For FML32, a fielded buffer pointer is of type `FBFR32 *`, a field length has the type `FLDLEN32`, and the number of occurrences of a field has the type `FLDOCC32`.

Fields are referred to by their field identifier in the FML32 interface. However, it is normally easier to remember a field name. There are two approaches to mapping field names to field identifiers. One is a compile-time mapping, the other is a run-time mapping.

Mapping Field Names to Field Identifiers

To avoid naming conflicts, BEA eLink Adapters must use the following run-time mapping method. Field name/identifier mappings can be made available to **FML32** programs at run-time through field table files. Field data types must be specifiable within field table files.

The FML32 interface uses the environment variables, `FLDTBLDIR32` to specify a list of directories where field tables can be found and `FIELDTBLS32` to specify a list of the files that are to be used from the table directories.

Note: The environment variables, `FLDTBLDIR32` and `FIELDTBLS32,` must be set prior to using FML32.

Within application programs, the FML32 function, `Fldid32,` provides for a run-time translation of a field name to its field identifier, and `Fname32` translates a field identifier to its field name. Type conversion should be performed implicitly via FML library functions. Implicit type conversion facilitates component reuse.

Use FML32 symbolic names and retrieve their values using `FLDID32`. The `Mkfldhdr32` function must not be used to build the adapter because it may cause conflicts with other field IDs.

Any field in a fielded buffer can occur more than once. Many FML32 functions take an argument that specifies which occurrence of a field is to be retrieved or modified. If a field occurs more than once, the first occurrence is numbered 0 and additional occurrences are numbered sequentially. The set of all occurrences make up a logical sequence, but no overhead is associated with the occurrence number (that is, it is not stored in the fielded buffer). If another occurrence of a field is added, it is added at the end of the set and is referred to as the next higher occurrence. When an occurrence other than the highest is deleted, all higher occurrences of the field are shifted down by one (for example, occurrence 6 becomes occurrence 5, 5 becomes 4, etc.).

Creating Field Names

Wherever possible, field names should match application field names one-to-one. Equivalent field names make the configuration easier to understand and manage. Field names must follow the convention of using only uppercase alphanumeric characters and underscores. This is a requirement of the Business Process Option (BPO).

Adapter-specific fields must not be required; they must be optional. Optional adapter-specific fields allow component reuse. Adapter-specific fields break the business process abstraction, requiring the designer of a process flow to have an understanding of the specific adapter for a given process step. Wherever possible, all adapter-specific field names should be configurable to avoid conflicts with other field names. Adapters should use a prefixing convention for any adapter-specific field names to avoid conflicts with other field names.

ud32 Client

There is an eLink-supplied client, *ud32*, that reads a tab delimited text file and uses the information from the file to construct an FML32 buffer. The ud32 client sends the buffer to a **service** that the user designates in the text file. ud32 is useful for testing. For an example of a text-delimited file that ud32 could use as input, see the "ud, ud32, wud, wud32" section in Appendix D, "Tuxedo Commands."

Refer to the "Example of a Server that Uses FML32" section in Appendix C, "FML32 API," for an example of simple code for a **server** that uses FML32.

FML32 Primitives

Following is a summary of some of the FML32 primitives that are used for all eLink programs including general eLink services and adapters. This subset of FML32 primitives should be sufficient to create most adapters. For more complete details and code examples, see Appendix C, "FML32 API," and the *BEA Tuxedo Reference Guide* at http://edocs.beasys.com/tuxedo/tux65/index.htm

Table 2-2 FML32 Primitives

FML Primitive	Description
`Fadd32`	Add new field occurrence
`Fchg32`	Change field occurrence value
`Ffind32`	Find field occurrence in buffer
`Fget32`	Get copy and length of field occurrence
`Fielded32`	Return true if buffer is fielded
`Finit32`	Initialize fielded buffer
`Fldid32`	Map field name to field identifier
`Fneeded32`	Compute size needed for buffer
`Fsizeof32`	Returns the size of an FML32 buffer

Warning: The `Falloc` function allocates FML buffers; however, buffers allocated using `Falloc` cannot be passed in a `tpcall`. **FML32** buffers that will be passed using the tpcall or tpacall **ATMI** primitives should be allocated by using a tpalloc with type parameter set to `FML32`.

Use FML32 symbolic names and retrieve their values using `Fldid32`. Field IDs must be determined dynamically at runtime or during initialization at boot time. The `Mkfldhdr32` function must not be used to build the adapter because it may cause conflicts with other field IDs.

eLink Commands

Commands are used to configure and administer the eLink runtime environment. Refer to *Administering the BEA Tuxedo System* for procedures and administrative tasks that are based on the command-line interface. For details about individual commands, refer to the *BEA Tuxedo Reference Manual*. Both documents may be found online at http://edocs.beasys.com/tuxedo/tux65/index.htm

Commonly Used Tuxedo Commands

Following is a list of the **Tuxedo** commands that are most commonly used for adapters. For complete details and code examples for each of the following commands, refer to Appendix D, “Tuxedo Commands.”

Table 2-3 Commonly Used Tuxedo Commands

Tuxedo Commands	Description
`buildclient`	Constructs a BEA Tuxedo client module. This command combines the files supplied by the -f and -l options with the standard BEA Tuxedo libraries to form a load module and invokes the platform's default compiler to perform the build.
`buildserver`	Constructs a BEA Tuxedo server load module. This command generates a stub file containing a main() function and invokes the platform's default compiler to perform the build.
`tmadmin`	Invokes the BEA Tuxedo bulletin board command interpreter. Refer to the “Commonly Used tmadmin Commands” section for more information.
`tmboot`	Invokes a BEA Tuxedo application with a configuration defined by the options specified.
`tmloadcf`	Parses a `UBBCONFIG` file and load binary `TUXCONFIG` configuration file.

Table 2-3 Commonly Used Tuxedo Commands

Tuxedo Commands	Description
`tmshutdown`	Shuts down a set of BEA Tuxedo servers.
`ud32`	Runs the BEA Tuxedo ud32 client that reads a tab delimited text file, produces an FML32 buffer, and uses the buffer to make a request to a specified service.

Commonly Used tmadmin Commands

The `tmadmin` command allows you to inspect and dynamically configure your eLink application. There are many commands that can be invoked from `tmadmin`, probably the most important being help. Several of the most useful commands are summarized in the following table

Table 2-4 Commonly Used tmadmin Commands

Command	Description
`help`	Prints help messages.
`quit`	Terminates the session.
`pclt`	Prints information for the specified set of client processes.
`psr`	Prints information for application and administrative servers.
`psc`	Prints information for application and administrative services.
`susp`	Suspends services.

For details about tmadmin commands, refer to the *BEA Tuxedo Reference Manual* at http://edocs.beasys.com/tuxedo/tux65/index.htm.

Hardware Requirements for the eLink Platform

The ADK hardware requirements are dictated by eLink adapter requirements. The MS VC++ v5.x command line compiler should be used for NT platforms.

Special Instructions for Installing the Tuxedo Core

For NT, use the InstallShield installer. For Unix, use the install.sh Korn shell script in the top level of the CD directory structure. Follow the instructions in the *BEA Tuxedo Installation Guide*, but pay special attention to the following tips.

- The default path for installation on Windows NT is under the Program Files directory. Using a directory with spaces in the name has presented a problem in prior releases of some of the Tuxedo utilities. Using spaces in the path directories should not be a problem with Tuxedo 6.5, but if you experience problems, choose a path name without spaces for installation.

  Note: To be MS Windows 2000 compliant, all programs must be installed in the Program Files directory.

- The **Web GUI** is not required.
- ***Important!*** When InstallShield asks if you want to install your Tuxedo license file at this time, answer **NO**. See the following section for instructions on preparing the license file.

Preparing the License File

In order to use the ADK you need a license for both the eLink Platform and the ADK. The license files are delivered on a floppy disk and should have accompanied your order of the eLink Platform and ADK. To use the license files, perform the following steps:

1. Copy the `LIC.TXT` file into the `TUXDIR/udataobj` directory.
2. This license file must have sections for [Tuxedo `6.5]`, `[eLink Platform]`, and `[eLink Adapter Development Kit]`. Refer to your Tuxedo documentation at http://edocs.beasys.com/tuxedo/tux65/index.htm for information about the Tuxedo license file.

Note: If you previously installed the eLink Platform the content of the supplied license files should be added to the LIC.TXT.

About the Tuxedo Simple Application

The Tuxedo Simple Application provides a way to verify your installation of Tuxedo as well as provide a working example of a Tuxedo application. A more comprehensive description of the Simple Application can be found in the *Tuxedo Application Development Guide*. The information provided here is specific to running the Tuxedo Simple Application in preparation for using the ADK.

All necessary code is located in the directory, `$TUXDIR\apps\simpapp`, where `$TUXDIR` is the directory where **Tuxedo** is installed. The Simple Application consists of a single **server** offering a single **service**. The service is called `TOUPPER`. You run the client with a single argument, which is a string to convert to upper case. The client calls the service, which returns the converted string. The client then prints the string.

Example: `simpcl “Hello World”`

Returned string is**:** `HELLO WORLD`

The Simple Application is designed so that it can be running within minutes after installing the Tuxedo software. You should probably copy the `simpapp` files to your own directory, since the configuration file must be edited and you might also want to experiment with the client and server code. Refer to the "Running a Sample Application" section of the *BEA Tuxedo Installation Guide* at http://edocs.beasys.com/tuxedo/tux65/index.htm for more information about running the Simple Application.

Setting Environment Variables

You need to set several environment variables before using the eLink Platform, running any eLink adapters, or running the Simple Application. Refer to the "Setting Up Your Environment" section of the *BEA Tuxedo Installation Guide* at http://edocs.beasys.com/tuxedo/tux65/index.htm for more information about setting environment variables. The following table lists the environment variables you need to set for any eLink Adapter.

Table 2-5 Environment Variables Used By the ADK

Variable Name	Description
`TUXDIR`	Base directory of the Tuxedo software.
`APPDIR`	Base directory for applications such as the sample program.
`PATH`	Must include `$TUXDIR/bin`.
`TUXCONFIG`	Full pathname of binary `tuxconfig` file.
`LD_LIBRARY_PATH`	Must include `$TUXDIR/lib` on systems that use shared libraries (except HP-UX and AIX).
`SHLIB_PATH`	*HP-UX only* - Must include `$TUXDIR/lib`.
`LIBPATH`	*AIX only* - Must include `$TUXDIR/lib`.

Following is an example of a script for Windows NT that could be used to set system variables:

Listing 2-1 Script for Setting System Variables for Windows NT

```
set TUXDIR=C:\tuxedo
set APPDIR=C:\simpapp
set PATH=%TUXDIR%\bin;%APPDIR%;%PATH%
set TUXCONFIG=%APPDIR%\tuxconfig
```

You must also set other environment variables if you are using a Workstation client.

Scripts are often used to save time in setting environment variables in a development directory. The following scripts are provided to set these variables for you, but the scripts must be edited for your environment.

- `$TUXDIR\apps\simpapp\setenv.cmd` for NT
- `$TUXDIR/tux.env for Unix`

Configuring the Simple Application

You configure the Simple Application by editing the sample configuration file, `ubbsimple`, and then submitting the `tmloadcf` command to create a binary configuration file for Tuxedo `tuxconfig`.

To configure the Simple Application, perform the following steps:

1. Edit the sample configuration file, `ubbsimple`, to replace the bracketed items with values appropriate to your installation.

   Note: Your `TUXDIR` and `TUXCONFIG` environment variables must match the values in the configuration file.

2. After editing `ubbsimple`, create the binary `TUXCONFIG` file with the command:

   ```
   tmloadcf ubbsimple
   ```

3. Answer **y** if you are asked whether to proceed.

   Note: After you create the initial `TUXCONFIG` file, you may recreate it using a `-y` command line option with `tmloadcf` to suppress the prompt.
   For example:

```
         tmloadcf -y ubbsimple
```

eLink creates a log file called `ULOG.mmddyy`. The default directory where this log resides is the application directory, `$APPDIR`. The creation of the binary file, `TUXCONFIG`, is logged in the `ULOG`. Any time there is some type of error or failure in your eLink adapter or eLink environment, the `ULOG` is one of the first places you should look.

Building the Client and Server for the Simple Application

For UNIX Operating Systems

The client and server for the Simple Application are already built. The executables are named *simpcl* and *simpserv*.

Or you can also build the client and server yourself by entering the following commands:

```
buildclient -o simpcl -f simpcl.c
buildserver -o simpserv -f simpserv.c -s TOUPPER
```

For Windows NT Operating Systems

You can build the server and client executables using the `makefile` `simpapp.nt`.

Or you can use the **`buildclient`** and **`buildserver`** commands described for UNIX operating systems in the previous section.

Booting the Simple Application

The Simple Application can be booted with the following command:

```
tmboot -y
```

After booting the system examine the `ULOG`. You will see a welcome message that is printed to the `ULOG` when `simpserv` is booted.

Then you can run `simpcl` as described in the "About the Tuxedo Simple Application" section above. `simpcl` can be run as many times as you wish.

Use the administrative command, `tmadmin`, to display and modify the parameters of the running application. For example, try any of the following parameters:

Parameter	Description
`psr`	**Printserver**-Print information for application and administrative servers.
`psc`	**Printservice**-Print information for application and administrative services.
`susp`	**Suspend**-Suspend services.

In particular, try suspending the `TOUPPER` service and then running the client. Refer to the *BEA Tuxedo Reference Manual* at http://edocs.beasys.com/tuxedo/tux65/index.htm for information about using commands.

Shutting Down the Simple Application

When you are done, you can shut down the Simple Application with the following command:

```
tmshutdown -y
```

Examine the `ULOG` after shutdown to review the messages that result from a shut down.

CHAPTER

3 Understanding Adapter Architecture and Design

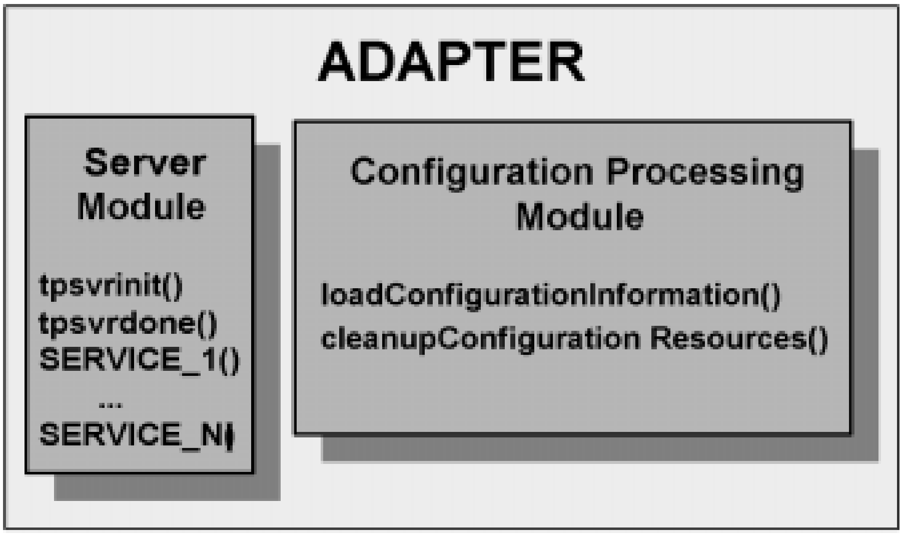
This chapter discusses the following topics:

- eLink Adapter Architecture Overview
- Application to eLink Adapters
- eLink to Application Adapters
- eLink Adapter Configuration
- Error Handling
- Deployment and Installation of eLink Adapters

eLink Adapter Architecture Overview

The standard design for an eLink adapter consists of at least two distinct program modules, the Server Module and the Configuration Processing Module.

Figure 3-1 Standard Adapter Design


ADAPTER
Server Module
tpsvrinit()
tpsvrdone()
SERVICE_1()
...
SERVICE_N()
Configuration Processing Module
loadConfigurationInformation()
cleanupConfiguration Resources()

The Server Module

The Server Module contains the code for the functions that perform the **services** advertised by the **server**. The Server Module also contains the code for the `tpsvrinit` function, which is called when the server is booted, and the `tpsvrdone` function, which is called at server shutdown time.

The `tpsvrinit` function performs the following tasks:

- Checks the license with the `eLA_chkeLinkLic` function in the ADK library.

  Warning! `eLA_chkeLinkLic` **must** be called to verify the eLink platform release. Failure to do so may result in incorrect runtime behavior in the adapter.

- Opens the message catalog.
- Retrieves the name of the configuration file.
- Reads the trace level from the configuration file.
- Calls functions from the Configuration Processing Module to make use of ADK API functions for parsing, processing, and storing the configuration data.

- Advertises services with the **ATMI** `tpadvertise` function.

Functions in the ADK library or the ATMI API facilitate most of these tasks.

The following table provides descriptions of the calls made by the `tpsvrinit` function.

Table 3-1 tpsvrinit Function Calls

tpsvrinit Tasks	**Description**
Check the license	Call `eLA_chkeLinkLic()`
Open catalog file	Call `eLA_OpenCatalogFile()`
Get configuration file	Call `eLA_GetConfigFileName()`
Read trace level	Call `eLA_SetServerMsgLevel()`
Parse, process, and store configuration data	Call `loadConfigurationInformation` and `cleanupConfigurationResources`.
Advertise **services**	Call `tpadvertise`

The Configuration Processing Module

Parsing and processing of the configuration file is done by the `loadConfigurationInformation` and `cleanupConfigurationResources` functions in the Configuration Processing Module. These functions make use of the ADK API functions for parsing, processing, and storing the configuration data.

Adapter Design Pseudo Code

Following is an example of a pseudo code outline that gives an overview of the general design of an adapter. The pseudo code outline follows these conventions:

- Comments, preceded by /* and followed by */, explain the purpose of particular sections of code.
- Functions from ATMI begin with the prefix `tp`.

- Functions from the ADK begin with the prefix `eLA_`.
- Text that is not commented or the name of a function from ATMI or ADK represents code that must be inserted to complete a required task.

In the pseudo code example, the server module has functions that support N services named `SERVICE_1` through `SERVICE_N`; however, `SERVICE_1` is the only service with any example code given.

Note: For an example of the general form of an application to eLink adapter that uses an FML32 buffer, see the sample application to eLink adapter that is shipped with the kit. However, the sample application to eLink adapter is not a complete example because it does not use a configuration file or the ADK library. For a complete working example see the sample eLink to application adapter code that is installed with the ADK.

Listing 3-1 Server Module Pseudo Code

```
/* tpsvrinit is called when a server is booted */
tpsvrinit( )
{
  /* Check license */
  eLA_chkeLinkLic( )
  /* Open the message catalog file. */
  eLA_OpenCatalogFile()
  /* Get the configuration file name. */
  eLA_GetConfigFileName()
  /* Read the trace level. */
  eLA_SetServerMsgLevel()
  /* Process configuration file      */
  loadConfigurationInformation( ) /* See configuration module below. */

  /* Call tpadvertise for each service name */
  for(i=0; i < numServices; i++)
  {
    tpadvertise( ) ;
  }
}

/* tpsvrdone is called at server shutdown time */
tpsvrdone( )
{
  /* Close the message catalog file. */
  eLA_CloseCatalogFile()
  cleanupConfigurationResources( ) /* See configuration module below. */
```

```
}

/*
** Supply functions that perform the actual services available to and or
** requested by the client. The argument to each is a structure TPSVCINFO
** (see definition following the code) containing, among other things a pointer
** to the data buffer, and the length of the data buffer.
*/

SERVICE_1(TPSVCINFO * request)
{
/* Check the type of the data member of the incoming TPSVCINFO struct */
  tptypes( )
```

Insert code to perform the actual **service** here. This will probably involve interacting with the third-party API in some way.

```
/* Return the transformed buffer to the requestor. */
  tpreturn( )
}
....
....
....
SERVICE_N(TPSVCINFO * rqst)
{
}
/* End of Server module */
```

Listing 3-2 Configuration Processing Module Pseudo Code

```
/* The following function is called in tpsvrinit to process the configuration file
information */
loadConfigurationInformation( )
{
  /* Allocate hash table for service information */
  eLA_InitHashtable( )

  /* Open Configuration */
  eLA_OpenTagFile( )

  /* Parse SERVER section, insure proper configuration file */
  eLA_GetFirstSection( )
  while( ADK_SUCCESS )
  {
    /* Parse properties for each SERVER section */
    eLA_GetFirstProperty( )
      while( ADK_SUCCESS )
      {
      eLA_GetNextProperty(  )
      }
    eLA_GetNextSection(sHandle)
  }
  eLA_CloseTagHandle( )

    /* Parse SERVICE(s) data for this SERVER */
    eLA_GetFirstSection( )
    /* Begin Services loop */
    while( ADK_SUCCESS )
    {
    /* Get first property for this service */
    eLA_GetFirstProperty( )


    /* begin Property loop */
    while( ADK_SUCCESS )
      {
```

Insert code here to process properties for each **service**.

```
      eLA_GetNextProperty(  )
      } /* end Property loop */

    /* Add service data element to hash table */
    eLA_put( )

    eLA_GetNextSection( )

    } /* End Services loop */

  eLA_CloseTagFile( )
  eLA_CloseTagHandle( )

}
/* This function was called in tpsvrdone to clean up any allocated resources. */
cleanupConfigurationResources( )
{
  /* Deallocate the hashtable allocated in loadConfigurationInformation */
  eLA_DestHashtable( )
}
```

The TPSVCINFO Structure

As shown in the SEVICE_1 and SERVICE_N lines of the above pseudo code, the argument to a function that performs an eLink **service** is a structure called `TPSVCINFO`. `TPSVCINFO` contains, among other things, a pointer to the data buffer, and the length of the data buffer. The full definition is given below:

Listing 3-3

```
Struct tpsvcinfo {

       char       name[32];     /* Service name */
       long       flags;        /* Options about the request */
       char       *data;        /* Request data */
       long       len;          /* Request data length */
       int        cd;           /* Connection descriptor */
       long       appkey;       /* Application key */
       CLIENTID   cltid;        /* Client identifier */
};
typedef struct tpsvcinfo TPSVCINFO;
```

Application to eLink Adapters

Application to eLink adapters translate requests initiated by the third-party software (for example, SAP, Broadvision, or MQseries) into standard eLink **ATMI** calls, such as a `tpcall`. The application to eLink adapter is an eLink **server** that will probably run as a “daemon” process. You can run the server as a daemon process by making a `tpacall` with the `TPNOREPLY` flag set to the daemon **service** from the server's tpsvrinit function. The daemon process starts when the server is booted. The code for the daemon service contains an infinite loop. Inside the loop, the server checks for input from the third-party software.

The method of communication between the third-party software and the adapter depends on the third-party interface. For example, for an MQSeries adapter, requests are written to a queue by MQSeries and the adapter monitors that queue. The eLink Platform routes the request. The Bulletin Board Liaison (**BBL**) consults the Bulletin

Board to find the name of the server that has advertised the requested service. The BBL then resolves the service name into a fully qualified address and sends the request to the server. The server or other adapter that advertises the desired service must be provided by the customer and is not part of the application to eLink adapter. However, in order to test an adapter, the adapter must be able to call an eLink service, so an eLink server that processes such a request should be part of a test for the adapter.

Note: All communication between the originating adapter and the adapter that services the request should be in FML32.

Following is an example of the server code that is specific to application to eLink adapters.

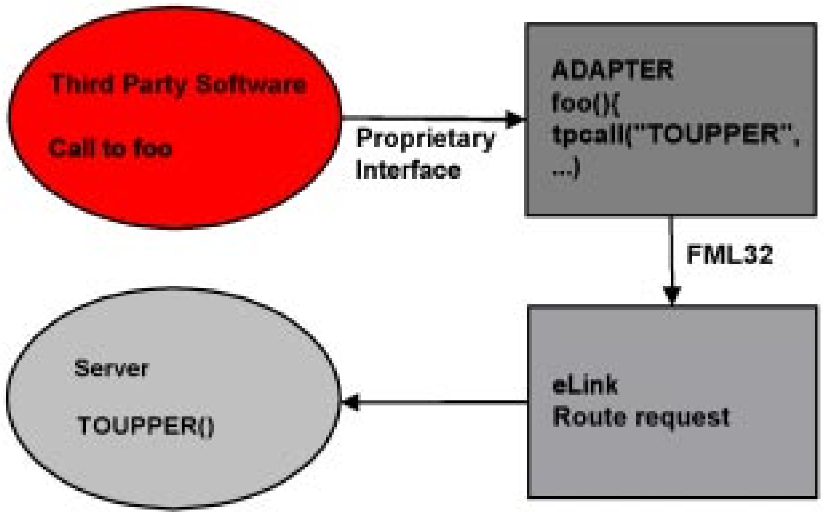
Listing 3-4 Server Module Code

```
tpsvrinit(argc,argv){
....
tpacall(“DAEMON”,...,TPNOREPLY);
return(0);
}

DAEMON (...){
....
for(;;)
  {
    //Check for requests from third-party application
  }
....
}
```

Following is an illustration of the typical request path for an application to eLink adapter.

Figure 3-2 Typical Request Path for an Application to eLink Adapter



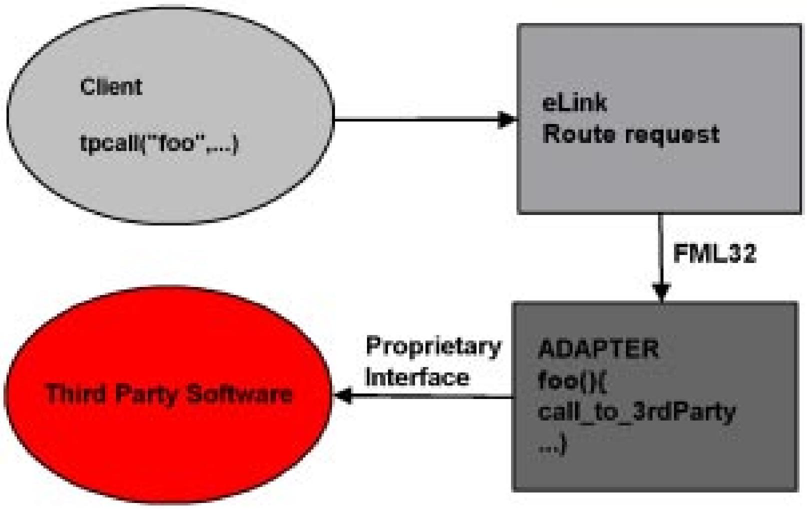
eLink to Application Adapters

eLink to application adapters translate requests made by requesting applications into calls to the API of third-party software. The eLink to application adapter is a **server**. A `tpcall` (or `tpacall`) initiates the request. The service name is defined by a parameter in the `tpcall`. The eLink Platform makes an association between the service name and a server, or adapter, that advertises that **service**. The service, a function defined in the adapter, makes a call to the third-party API. The interface between the adapter and the third-party software is defined by the third-party software.

The sample eLink to application adapter included with the ADK is a complete working example of an eLink to application adapter. Client code is provided to initiate the request. In real environments this originating call might be from another adapter or the Business Process Option. This code is not part of the adapter, however such a client should be written to test the adapter.

Following is an illustration of the typical request path for an eLink to application adapter.

Figure 3-3 Typical Request Path for an eLink to Application Adapter



eLink Adapter Configuration

The eLink Adapter configuration is defined in the SERVERS section of the **`UBBCONFIG`** file. The `UBBCONFIG` file is located in the directory specified by the configuration. You must create a custom `UBBCONFIG` file and add the configuration information for the eLink Adapter to the SERVERS section.

The method of configuring adapters and how the adapter processes that configuration, follows a standard format. All adapter code and any test configurations must follow this standard.

Standards for Adding an eLink Adapter to the UBBCONFIG File

In a `UBBCONFIG` file, lines beginning with an asterisk (*) indicate the beginning of a specific section. The name of the section immediately follows the *. The beginning of the SERVERS section is marked *SERVERS. Parameters are generally specified by `KEYWORD= value`. Entries in the SERVERS section have the form:

```
AOUT  required parameters  [optional parameters]
```

where `AOUT` specifies the file (string_value) to be executed by `tmboot`.

Required parameters are:

```
SRVGRP = string_value
```

which specifies the name for the group in which the server is to run. The `string_value` must be the logical name associated with a server group in the GROUPS section. The `string_value` must be 30 characters or less. This association with an entry in the GROUPS section means that `AOUT` is executed on the machine with the Logical Machine ID (`LMID`) specified for the server group. The GROUPS section also specifies the `GRPNO` for the server group and parameters to pass when the associated resource manager is opened. All server entries must have a server group parameter specified:

```
SRVID = number
```

which specifies an integer that uniquely identifies a **server** within a group. Identifiers must be between 1 and 30,000 inclusive. This parameter must be present on every **server** entry.

All adapter entries in the `UBBCONFIG` file are required to have the **`CLOPT`** parameter (although it is listed as optional in the Tuxedo online documentation.) **`CLOPT`** specifies **`servopts`** options to be passed to the **server** when booted. “--” marks the end of system-recognized arguments and the start of arguments to be passed to a subroutine within the server. All eLink adapters will have exactly one argument after the --, the `-c` option followed by the name of the adapter-specific configuration file. All other configuration parameters specific to the adapter should appear in the adapter-specific configuration file. For more details on the `CLOPT` parameter see Appendix E, “Servopts.”

Listing 3-5 Example of the CLOPT Parameter Entry

```
*SERVERS
elinkmqi
SRVID=”number”
REPLYQ=N
CLOPT="--  -C configuration_file_name"
```

Sample UBBCONFIG File

Listing 3-6 Sample UBBCONFIG File

```
*RESOURCES

IPCKEY   123791
DOMAINID simpapp
MASTER   simple

*MACHINES

DALNT6
       LMID= simple
       TUXDIR= "\tuxedo"
       TUXCONFIG= "\myappdir\tuxconfig"
       APPDIR= "\myappdir"
       FIELDTBL32= "sample.fml"
       FLDTBLDIR32= "\myappdir"
       ULOGPFX= "\myappdir\ULOG"


*GROUPS

eLINK
         LMID=simple  GRPNO=1

*SERVERS
DEFAULT:
         CLOPT="-A"

elinkmqi
              SRVGRP=eLINK
              SRVID=10
              CLOPT="-- -C config.file"

*SERVICES

*ROUTING
```

eLink Adapter Configuration Files

In addition to the configuration information contained in the application’s **`UBBCONFIG`** file, each adapter must have its own specific configuration file. You must create the configuration file following the guidelines specific to the type of adapter.

The adapter configuration file defines aliasing of service names, tracing parameters, and the names of outside resources to be used. Aliasing allows the names of advertised **services** to be related to the **business logic** and also allows for many service names to be mapped to a single implementation. In the configuration file, users can activate tracing and specify the level of tracing that is done. If a **server** needs to retrieve information from outside resources, for example read requests from a queue, the names of the outside resources are specified in the configuration file.

The eLink Adapter reads the configuration file at startup. The configuration file is an ASCII text file that the user creates for the adapter. The user arbitrarily chooses the name of this file, but it must match what is specified by the **`CLOPT`** parameter in the `UBBCONFIG` file (as described in “Standards for Adding an eLink Adapter to the UBBCONFIG File” on page 3-11). This configuration file must be located in the application directory (`APPDIR`) for the end-user's application.

Structure of the eLink Adapter Configuration File

The adapter configuration file is divided into several sections. Each adapter configuration file contains one, and only one, SERVER section. The configuration file may also contain one or more SERVICE sections and one or more FIELDMAP sections.

When you create an eLink Adapter configuration file, some standard conventions apply to the format. Following are the standard conventions that should be used for the adapter configuration file:

- The name of a section (e.g., SERVER or SERVICE) is always preceded by an asterisk (*).
- The section name follows the asterisk and is entered in all upper case.
- The parameter names and values appear on the lines following the section names.
- The parameter name is entered in uppercase followed by an equal sign and the parameter value. If there is no parameter value following the equal sign, an error is returned, but the `Tag Value` field is filled in and the configuration file is still processed.
- A # sign is treated as the beginning of a comment UNLESS it is preceded by a backslash. If your data entry requires a # sign, use a backslash as in the following example:

  `Phone \#-800.555.1212` returns `Phone #-800.555.1212`

Example of the format of a configuration file section and parameter:

```
*<NAME OF SECTION>

<PARAMETER NAME>=<PARAMETER VALUE>
```

The SERVER Section

There is only one SERVER section in each configuration file. In this section, the only required parameters are the `MAXMSGLEVEL` and `MINMSGLEVEL` tracing level parameters. The following table provides descriptions for these parameters:

Parameter Name	Description
`MAXMSGLEVEL=<int>`	Indicates the maximum level of tracing messages the adapter is to log.
`MINMSGLEVEL=<int>`	Indicates the minimum level of tracing messages the adapter is to log.

For more information about the trace levels, refer to the "Tracing" section in this chapter.

The SERVICE Section

You must define the **services** that will be advertised in the SERVICE sections of the configuration file. Each defined service has a corresponding section in the configuration file. Each defined service section has exactly one required parameter, `NAME`. The `NAME` parameter must be 15 characters or less in length. The following table provides a description for this parameter.

Parameter Name	Description
`NAME=<Name>`	Defines the eLink **service** name that is to be advertised.

The SERVICE section is where names advertised by eLink and related to business logic are mapped to the names of the actual functions that perform the services.

For example, the eLink Adapter for XML performs data format conversion between FML and XML data formats. For each conversion, you must create a SERVICE section in the adapter configuration file. This SERVICE section describes an eLink service that the adapter advertises in order to perform a conversion. End-user applications then request this service to perform the necessary conversion.

The SERVICE definition maps an eLink service name to a specific type of conversion. The service name is arbitrarily chosen. These SERVICE definitions allow different conversions to be represented by different eLink service names. For example, you could define services `CONVERT_A`, `MAKEXML`, or `MYFMLXML` that are all FML to XML conversions. This allows many conversions to be mapped to one implementation. However, these SERVICE names cannot equal the `CONVERSION` type parameter.

The FIELDMAP Section

A field map is a set of mappings of the names of fields used by the third-party software to the names of the corresponding FML32 fields. A field map must be defined in a section called FIELDMAP. The first parameter of the FIELDMAP section must be `FMID` (field map identifier). This parameter identifies a field map and is referenced by the service definition using the map. A field map may be referenced by more than one **service**, if applicable.

The following format is used in adapter configuration files to define each mapping in a field map:

```
Application Name:FML32 Field Name:input/output:field designator
```

where

`Application Name` is the name of the application field

`FML32 Field Name` is the name of the FML32 field associated with this Application field name.

`Input/output` defines whether a field is expected as input or passed as output or both. Valid values are `I, O,` and `IO`.

`Field designator` is an adapter-defined designator for the defined field. This parameter can be used to designate required fields, key fields, optional fields, etc.

```
R = Required Field
O = Optional Field
K = Key Field
P = Parent Key Field
L = Link Field
G = Group Field
```

Additional values can be defined by the adapter, if necessary

Example:

```
*SERVICE
NAME=NwCont
BUSINESS_OBJECT=Account
BUSINESS_COMPONENT=Contact
OPERATION=NEW
FMID=Map1

*FIELDMAP
FMID=Map1
Birth Date:EL_SBL_BIRTH_DATE:I:O
Comment:EL_SBL_COMMENT:I:O
Credit Agency:EL_SBL_CREDIT_AGNCY:I:O
Credit Score:EL_SBL_CREDIT_SCORE:I:O
Email:EL_SBL_EMAIL:I:O
Address:EL_SBL_ADDRESS:I:O
Job Title:EL_SBL_JOB_TITLE:I:O
First Name:EL_SBL_FIRST_NAME:I:R
Last Name:EL_SBL_LAST_NAME:I:R
Id:EL_SBL_ID:O:R
```

In the above example, the application field "`Birth Date`" maps to the FML32 field `EL_EBL_BIRTH_DATE`. This field is an optional (designated by an `O`) input (indicated by the I/O type of `I`) field. The application field "`ID`" is mapped to `EL_SBL_ID`. This field is defined as a required output field.

The ADK contains the following functions that are used to parse the field map sections in the adapter configuration file: `eLA_GetFieldMap`, `eLA_GetFirstField`, `eLA_GetNextField`. For complete details and code examples see the “Configuration Processing API” section in Appendix A.

Sample Adapter Configuration File

The following is an example of a configuration file for an adapter that difines one service.

Listing 3-7 Adapter Configuration File

```
# This is a comment

*SERVER
MAXMSGLEVEL=10
MINMSGLEVEL=0
*SERVICE
NAME=CONVERT_WITHDRAWAL
CONVERSION_TYPE=FMLMTI2XML
MTI_NAME=withdraw.mti

*SERVICE
NAME=CONVERT_DEPOSIT
CONVERSION_TYPE=FML2XML
LIST_TAG_SUFFIX=_LIST
FMID=Map1
*FIELDMAP
FMID=Map1
Birth Date:EL_SBL_BIRTH_DATE:I:O
Comment:EL_SBL_COMMENT:I:O
Credit Agency:EL_SBL_CREDIT_AGNCY:I:O
Credit Score:EL_SBL_CREDIT_SCORE:I:O
Email:EL_SBL_EMAIL:I:O
Address:EL_SBL_ADDRESS:I:O
Job Title:EL_SBL_JOB_TITLE:I:O
First Name:EL_SBL_FIRST_NAME:I:R
Last Name:EL_SBL_LAST_NAME:I:R
Id:EL_SBL_ID:O:R
```

API to Parse and Store Configuration Data

The ADK includes an API to facilitate the parsing of the Adapter configuration file and to store configuration information for quick lookup.

API to Parse the Configuration File

The following API functions may be used to facilitate the parsing of the Adapter configuration file. For complete details and code examples see the "Configuration Processing API" section in Appendix A, "eLink Adapter Development Kit References."

Table 3-2 Configuration Processing API Functions

Configuration Processing API Name	Description
`eLA_OpenTagFile`	Opens a Tag (or config.) file, and reads it into memory.
`eLA_CloseTagFile`	Closes a handle returned by `eLA_OpenTagFile`.
`eLA_CloseTagHandle`	Closes a handle returned by `eLA_GetFirstSection`.
`eLA_GetFirstSection`	Searches the config file memory image for desired section.
`eLA_GetNextSection`	Finds next occurrence of desired section.
`eLA_GetFirstProperty`	Retrieves Tag/Value pair for first property in a section.
`eLA_GetNextProperty`	Retrieves Tag/Value pair for successive properties in a section.
`eLA_GetPropertyValue`	Retrieves value for first occurrence of a tag in a section.
`eLA_GetFieldMap`	Searches for the named *FIELDMAP section.
`eLA_GetFirstField`	Retrieves the information for the first line in a fieldmap section.

Table 3-2 Configuration Processing API Functions

Configuration Processing API Name	Description
`eLA_GetNextField`	Retrieves the information for successive lines in a fieldmap section.

API to Store the Configuration Data

After the configuration information has been parsed, it can be stored in a hash table to facilitate quick lookup. The hash table API is included in the ADK. A summary of the available functions is provided here. For complete details and code examples see the "Hash Table API" section in Appendix A, "eLink Adapter Development Kit References."

Table 3-3 Hash Table API Functions

Hash Table API Name	Description
`eLA_InitHashTable`	Creates a hash table.
`eLA_DestHashTable`	Frees all dynamic memory in the hash table.
`eLA_put`	Adds a new element to the hash table.
`eLA_get`	Retrieves an element from the hash table.
`eLA_hash`	Returns the hash value for a given key.

Error Handling

Error handling may be accomplished through error logging and tracing. Error logging is mandatory and must always be "turned on", while tracing can be activated or deactivated by the user.

If an eLink to application adapter successfully completes a service request, then the adapter returns with `tpreturn` (`TPSUCCESS`, ….). The `tpurcode`, the second parameter in `tpreturn`, should be set to the value of the third-party API return code (if available and applicable). This is to ensure that other eLink components can determine that the application request has been successfully executed.

Errors from eLink to application adapters should return in a consistent manner. Consistency allows other eLink components, such as the **Data Integration Option** (DIO) or the **Business Process Option** (BPO), to detect and respond to errors.

eLink adapters need to handle two types of errors, business level exceptions and infrastructure errors. Proper error handling ensures that all the eLink components recognize error codes that are returned for eLink to application adapters.

Business Level Exceptions

Business level exceptions are those that occur when the advertised service is successfully invoked by the adapter, but the called application is unable to complete the requested operation. For example, if the business service advertised is "Ship Order", the service may fail if one of the items to be shipped is out of stock and the incomplete order may not be shipped. This exception must be returned to the caller but will not be logged.

If a business-level exception occurs, then the adapter returns with `tpreturn` (`TPFAIL`, `0`, ...). The details of the error, for example the application error code, are returned in the `ELINK_APP_ERR` FML32 field. This is a string field. Using the recommended `CFchg32()` call, the adapter may populate the `ELINK_APP_ERR` FML32 field with either an error number or error string without further conversion.

Infrastructure Level Exceptions

Infrastructure level exceptions are those in which the adapter encounters an uncorrectable error, for example, a failure to allocate an FML32 buffer or other memory allocation errors within the adapter code. All infrastructure level errors are returned to the caller and logged using the `eLA_log()` function that is included in the ADK. A message catalog should be used (See the following "Message Catalog" section). The `eLA_catentry` function is used to retrieve the actual message string from the catalog using a message number.

If an eLink to application adapter fails because of an infrastructure level error, then the adapter returns with `tpreturn (TPFAIL, !0, ...)`. The FML32 field, `ELINK_ADAPTER_ERR`, contains an error message. The category of adapter error is indicated in the FML32 `ELINK_ADAPTER_ERR_CODE` field. The content of this string field is a single keyword. A predefined set of categories is described in the list below. Whenever possible, errors should be mapped to these categories. Adapter authors may define additional categories, however, third-party additions should omit the "ELINK_" prefix.

Table 3-4 Adapter Error Categories

Category	Description
`ELINK_EAPP_API`	The application's API returned an error. Note that this refers to the application's API returning an infrastructure level error rather than a business level error.
`ELINK_EAPP_UNAVAIL`	The application was unavailable.
`ELINK_EATMI`	An `ATMI` error occurred.
`ELINK_ECONFIG`	An error occurred with the adapter configuration data.
`ELINK_EFML`	An FML error occurred.
`ELINK_EINVAL`	Invalid value/argument error. For example, an FML32 request buffer is sent to an adapter without all the required FML32 fields being present.
`ELINK_EITYPE`	An input type mismatch. For example, converting between FML32 and application data types on the input.
`ELINK_ELIMIT`	Out of range value.
`ELINK_ENOENT`	No entry found. The application functionality corresponding to the **service** could not be found.
`ELINK_EOS`	An operating system error. For example, a memory allocation error.
`ELINK_EOTYPE`	An output type mismatch. For example, converting between **FML32** and application data types on the output.
`ELINK_EPERM`	A permissions error.

Table 3-4 Adapter Error Categories

Category	Description
`ELINK_EPROTO`	A protocol error.
`ELINK_ETIME`	A timeout error. For example, timing-out while waiting for the application to process the request.
`ELINK_ETRAN`	A transaction error.

Message Catalog

All error and tracing messages should be put in a message catalog file. The catalog file is assumed to be a text file (.txt) whose message lines adhere to the rules outlined in the HP-UNIX gencat() MAN pages. The MAN pages provide reference information in an online format.

For example:

```
10 “WARN: Existing parameter %s = %d, cannot change to %d”
11 “ERROR: Memory allocation failure”
```

The message numbers should be in ascending order, but the numbers need not be contiguous. Double quotes are stripped and lines starting with comments (‘$’) are disregarded. Sets are not supported.

These catalog files are required to be located in the `$TUXDIR\ELINK\CATALOGS` directory. However, the ADK utility function that is used to open the catalog file, `eLA_OpenCatalogFile()`, expects a fully qualified path name.

In the event that the message number is not found in the catalog file, a string similar to `“Message xxx not found”` is returned to the caller.

API to Access the Message Catalog File

The ADK includes an API to access the message catalog file. A summary of the available functions is provided here. For complete details see the “Utility Functions and Macros” section in Appendix A, “eLink Adapter Development Kit References.”

Message Catalog API Name	**Description**
`eLA_OpenCatalogFile`	Open the message catalog file
`eLA_CloseCatalogFile`	Close the message catalog file
`eLA_catentry`	Retrieve the message corresponding to the entry number

The following code segment illustrates how these functions are used:

Listing 3-8 Code for Catalog File Functions

```
ADK_CAT_HANDLE rHandle;
char catFileName[MAX_FNAME];
char msgbuffer[1024];
....
rHandle = eLA_OpenCatalogFile(catFileName);
....
eLA_catentry(msgbuffer, sizeof(msgbuffer), rHandle, 11);
....
eLA_CloseCatalogFile(rHandle);
```

Tracing

eLink adapters should be written to allow tracing to be enabled through the adapter-specific configuration file. Tracing can then be activated or deactivated by the user.

Trace Levels

A trace level parameter is associated with each tracing message. The `MAXMSGLEVEL` and `MINMSGLEVEL` parameters are set in the adapter configuration file to specify the range of trace messages to be printed. The `MAXMSGLEVEL` and `MINMSGLEVEL` parameters are read in the `tpsvrinit` function by calling the eLA_Set**Server**MsgLevel function. They are then stored in the corresponding fields of a MSG_LEVEL structure. For more information, refer to the "Definitions and Typedefs" section in Appendix A, "eLink Adapter Development Kit References."

MAXMSGLEVEL and MINMSGLEVEL parameters have a range of values from 0 to 9. If both the MAXMSGLEVEL and MINMSGLEVEL parameters are set to 0, then no tracing is done. Following are guidelines for assigning a message level to each individual trace statement.

Table 3-5 Error Message Levels

Level Range	Corresponding Tracing
1-3	Minimal level of tracing. Log module, program, or major function entry points only.
4-6	Moderate level of tracing. Log entry into major control blocks or execution of key events in the program. Log exit points from modules, programs, and functions.
7-9	Very detailed level of tracing. All function calls and return codes are printed. All buffers are hex dumped. Entry and exit from all functions are logged.

If there is any code in the adapter that does signal or event handling, it is advisable to use a trace level in the 7-9 range so that the signal or event handling is turned off for the purposes of debugging. Actual usage of trace values should be described in user documentation for the adapter.

Tracing Functions and Macros

In addition to the eLA_log function, there are two other functions included in the ADK to help with tracing, eLA_hexdump and eLA_catentry. eLA_hexdump performs a formatted hexdump of a buffer. eLA_catentry retrieves an entry from the message catalog. The macro, ELACATENTRY, serves as a cover for the eLA_catentry function.

The ADK includes two macros, ELATRACE and ELAIFTRACE, that are specifically designed to aid in tracing. ELATRACE has three arguments: VAR, LVL and ARGS. If the LVL argument is between the values of the fields of the MSG_LEVEL structure VAR (inclusively), then the argument ARGS is substituted into an eLA_log function call. ELAIFTRACE has two arguments: VAR and LVL. ELAIFTRACE evaluates to an if statement that checks to see if `LVL` is between the values of the fields of the `MSG_LEVEL` structure `VAR`. `ELAIFTRACE` can be used to check if bracketed code immediately following it should be evaluated.

The following code segment illustrates the use of the tracing functions and macros:

Listing 3-9 Code for Tracing Functions and Macros

```
MSG_LEVEL zLevel = {0,0};
char configFileName[MAX_FNAME];
...

rc = eLA_GetConfigFileName(configFileName, MAX_FNAME, argc, argv);
printf("eLA_GetConfigFileName - rc = %d\n", rc);
if(rc == -1)
  printf("Buffer not large enough\n");
else if(rc == 0)
  printf("File Name parameter not found\n");
else
  printf("File Name = %s\n", configFileName);
...
rc = eLA_SetServerMsgLevel(configFileName, &zLevel);
printf("SetServerMsgLevel - rc = %d\n",rc);
if(rc == ADK_SUCCESS)
  printf("min, max = %d, %d\n", zLevel.minMsgLevel, zLevel.maxMsgLevel);
for(i = 0;i < 10;i ++)
  { ELAIFTRACE(zLevel, i) printf("Level = %d\n", i);}
for(i = 0;i < 10;i ++)
  ELATRACE(zLevel, i, ("Log msg level %d", i));
...
```

Deployment and Installation of eLink Adapters

Adapters should be compatible with all the following platforms supported by eLink Platform v1.1 or higher. Additional operating systems may be supported in future releases of the eLink Platform and the ADK.

- HP 10.20 and 11.00

- AIX 4.3.x
- Compaq Tru64 UNIX 5.0
- Solaris 2.6 and 7
- NT 4.0

For HP-UX builds, the `+DAportable` compilation flag should be used to make the resulting object files portable across PA-RISC 1.1 and 2.0 workstations. Other important build flags are `-Wl,+s`. This is actually a command for the linker to use the `SHLIB_PATH` environment variable to locate shared libraries.

As an example, `CFLAGS` should at least use the following parameters:

```
CFLAGS=+DAportable -Wl,+s
```

Installation Directory Structure for Components

Adapters are installed in the eLink Platform using the traditional configuration shown in the following table. Your code needs to work in this environment. In the table, *adapter* should be replaced by the name of your adapter (PeopleSoft, SAP, etc.).

Table 3-6 Directory Path for Tuxedo Components Relating to Adapters

Directory	Contents
`$(TUXDIR)/bin`	Executables and dynamic link libraries.
`$(TUXDIR)/lib`	Library objects (*.so, *.a, *.sl, *.lib).
`$(TUXDIR)/include`	Include files needed for customer applications.
`$(TUXDIR)/ELINK/CATALOGS`	Message catalog files.
`$(TUXDIR)/udataobj`	FML fldtbls for internal adapter use and binfiles.adapter.
`$(TUXDIR)/`*`adapter`*	Adapter-specific files and samples. It may contain subdirectories.

Table 3-6 Directory Path for Tuxedo Components Relating to Adapters

Directory	Contents
`$(TUXDIR)/`*`adapter`*`/mysample`	Sample applications provided with the adapter where *mysample* is the name of the sample application provided.

Each Adapter should provide a flat text file named binfiles.adapter for inclusion in the `$(TUXDIR)/udataobj` directory. The flat text file lists the files that are deliverables for this product. These deliverables should be placed in the following directory paths:

Table 3-7 Directory Path for Adapter Deliverables

Directory	Deliverables
`/(base directory for installation)`	All make, bat, script files and readme files.
`/doc`	All documentation.
`/src`	All source code.
`/include`	All include files.
`/bin`	All binary command line executables (if any).
`/dll`	All dlls (if any).
`/lib`	All libs (.SL, .SO, .LIB, etc.) (if any).
`/demo`	Any demo code, sample programs, etc.
`/test`	All test material (scripts, data, and programs).

Naming Convention for Source and Executable Files

The source and executable files for all adapters must follow specific naming conventions.

The naming convention for source files is:

```
eLink<abbeviation><i or o>.c
```

where <`abbreviation`> is the two or three letter abbreviation associated with the adapter and i is for inbound (application to eLink) and o is for outbound (eLink to application).

The naming convention for executable files is:

```
ELINK<abbreviation><I or O>
```

where <`abbreviation`> is the two or three letter abbreviation associated with the adapter (in caps this time) and I is for inbound (application to eLink) and O is for outbound (eLink to application). Using all caps for the names of **servers** is an eLink convention.

CHAPTER

4 Installing the eLink ADK and Sample Adapters

This chapter discusses the following topics:

- What is Included in the eLink ADK
  - Include Files and Libraries
  - The Sample Application to eLink Adapter
  - The Sample eLink to Application Adapter
  - The Sample E-Mail Adapter
- Installing the eLink Adapter Development Kit
  - Installing on the Windows NT Platform
  - Installing on the HP-UX, AIX, Solaris, and Compaq TRU64 UNIX Platforms

What is Included in the eLink ADK

Your eLink Adapter Development Kit is shipped with include files, shared libraries, and sample application to eLink and eLink to application adapters.

Note: In order to use the ADK you need a license for both the eLink Platform and the ADK. The license files are delivered on a floppy disk and should have accompanied your order of the eLink Platform and ADK. Refer to "Preparing the License File" on page 2-11 for more information about the license file.

Include Files and Libraries

In addition to the required include files (.H); the development kit includes shared libraries, (.sl) for HP-UX 10.20 and 11.0, (.so) for Solaris 2.6 and AIX 4.3.x; and a dynamic link library (.dll) and import library (.lib) for Windows NT. The .dll is compiled with Microsoft VC++ v5.x. While the .dll may be used with any Windows development platform, the import library is specific to the Microsoft compiler.

Note: The HP 10.20 operating system is not supported for this release of the Sample E-Mail Adapter because the SMTP library does not support HP 10.20.

Table 4-1 Shipping List for ADK Include Files and Libraries

File	Description
`adkdemo\adkdemo.text`	Sample message catalogue
`include\adkfns.h`	Include file for both NT and Unix
`include\adklog.h`	Include file for both NT and Unix
`include\adktypes.h`	Include file for both NT and Unix
`include\fmlfns.h`	Include file for both NT and Unix
`bin\libadk.dll`	Dynamic link library for Windows NT
`lib\libadk.lib`	Import library for Windows NT
`lib\libadk.sl`	Shared library for HP 10.20 or 11.00
`lib\libadk.so`	Shared library for Solaris 2.6 or 7 or Compaq Tru64 UNIX 5.0
`lib\libadk.a`	Shared library for AIX 4.3
`lib\libsmtp.sl`	Shared SMTP library for HP 11.00

Table 4-1 Shipping List for ADK Include Files and Libraries

File	Description
`lib\libsmtp.so`	Shared SMTP library for Solaris 2.6. or 7 or Compaq Tru64 UNIX 5.0
`lib\libsmtp_shr.a`	Shared SMTP library for AIX 4.3
`lib\libcomm.so`	Communication protocol library for Solaris 2.6. or 7 or Compaq Tru64 UNIX 5.0
`lib\libcomm_shr.a`	Communication protocol library for AIX 4.3
`lib\libcomm.sl`	Communication protocol library for HP 11.00
`lib\libcomm.dll`	Communication protocol dynamic link library for Windows NT
`lib\libcomm.lib`	Communication protocol library for Windows NT

The Sample Application to eLink Adapter

The sample application to eLink adapter contains code to generate the application to eLink adapter program, which is a Tuxedo **server**. The sample application to eLink adapter also contains a demo server program that can be used to test the adapter. The application to eLink adapter also includes a sample `UBB` file, as well as .bat and .SCR files to set up the environment. Since the .lib files included with **Tuxedo** are specific to the MS VC++ compiler, and **`BUILDSERVER`** and **`BUILDCLIENT`** default to calling the command line version of the MS compiler (CL), the .bat files were created for use with the MS compiler.

Table 4-2 Shipping List for Sample Application to eLink Adapter Components

File	Description
`adkdemo\inbound\elinkdi.mak`	.MAK file for Unix
`adkdemo\inbound\inbound.c`	Source code for the demo application to eLink server program for Unix

Table 4-2 Shipping List for Sample Application to eLink Adapter Components

File	Description
`adkdemo\inbound\input.txt`	File containing requests
`adkdemo\inbound\README`	README file for Sample Application to eLink Adapter
`adkdemo\inbound\SETENV.BAT`	Sets the various environment variables required by Tuxedo for NT
`adkdemo\inbound\setenv.sh`	Sets the various environment variables required by Tuxedo for Unix
`adkdemo\inbound\simpserv.c`	Source code for the Tuxedo server program
`adkdemo\inbound\table.fml`	FML32 table
`adkdemo\inbound\ubb.inbound`	UBBCONFIG file

The Sample eLink to Application Adapter

The sample eLink to application adapter contains code to generate demo server and client programs, sample .UBB and .CFG files, as well as .bat and .SCR files to set up the environment and automate the build process. Since the .lib files included with **Tuxedo** are specific to the MS VC++ compiler, and **`BUILDSERVER`** and **`BUILDCLIENT`** default to calling the command line version of the MS compiler (CL), the .bat files were created for use with the MS compiler.

Table 4-3 Shipping List for Sample eLink to Application Adapter Components

File	Description
`adkdemo\outbound\ADKDEMO.C`	Source code for the demo server program
`adkdemo\outbound\ADKDEMO.CFG`	CFG file
`adkdemo\outbound\ADKDEMO.H`	Source code for the demo server program
`adkdemo\outbound\ADKDEMO.UBB`	UBB file for NT

Table 4-3 Shipping List for Sample eLink to Application Adapter Components

File	Description
`adkdemo\outbound\ADKDEMOX.UBB`	UBB file for Unix
`adkdemo\outbound\CONFIG.C`	Routines to verify and load CFG file
`adkdemo\outbound\CONFIG.H`	Routines to verify and load CFG file
`adkdemo\outbound\democlient.C`	Source code for the demo client program
`adkdemo\outbound\elinkdo.mak`	.MAK file for Unix
`adkdemo\outbound\FOOFNS.C`	String handling functions to simulate the calling of an outside API
`adkdemo\outbound\FOOFNS.H`	String handling functions to simulate the calling of an outside API
`adkdemo\outbound\makeclient.BAT`	Makes the demo client program
`adkdemo\outbound\makeclient.SCR`	Makes the demo client program
`adkdemo\outbound\makeserver.BAT`	Makes the demo server program
`adkdemo\outbound\makeserver.SCR`	Makes the demo server program
`adkdemo\outbound\README`	README file for Sample eLink to Application Adapter
`adkdemo\outbound\SETADK.BAT`	Sets the various environment variables required by `Tuxedo`
`adkdemo\outbound\SETADK.SCR`	Sets the various environment variables required by `Tuxedo`

The Sample E-Mail Adapter

The sample E-Mail Adapter contains code to generate demo E-Mail server and client programs. This adapter is an eLink to Application adapter.

Table 4-4 Shipping List for Sample E-Mail Adapter

File	Description
adkdemo\email\config.c	Routines to verify and load CFG file
adkdemo\email\config.h	Routines to verify and load CFG file
adkdemo\email\emailclient.c	Source code for the email client program
adkdemo\email\emailserver.c	Source code for the email server program
adkdemo\email\emailserver.cfg	Configuration file for the server
adkdemo\email\emailserver.h	Header file for the email server program
adkdemo\email\emaildemo.ubb	UBB file for NT
adkdemo\email\emaildemox.ubb	UBB file for UNIX
adkdemo\email\emaildemo.mak	.MAK file for UNIX
adkdemo\email\emailtools.c	Source code used to configure and send email
adkdemo\email\emailtools.h	Header file for emailtools
adkdemo\email\email.cfg	Email configuration file (used with -f option)
adkdemo\email\emailmessage.txt	Sample email message to send
adkdemo\email\makeclient.bat	Makes the email client program (NT)
adkdemo\email\makeclient.scr	Makes the email client program (UNIX)
adkdemo\email\makeserver.bat	Makes the email server program (NT)
adkdemo\email\makeserver.bat	Makes the email server program (UNIX)
adkdemo\email\README.DEMO	README file for Sample Email Adapter
adkdemo\email\setadk.bat	Sets the various environment variables required by Tuxedo (NT)
adkdemo\email\setadk.scr	Sets the various environment variables required by Tuxedo (UNIX)

Installing the eLink Adapter Development Kit

BEA-branded adapters must be buildable on all supported eLink Platform operating systems. HP-UX 11.00 platforms use the standard HP C compiler and NT platforms use the MS VC++ v5.x command line compiler.

Complete the following tasks prior to installing the eLink Adapter Development Kit:

- Read the *`eLink Adapter Development Kit` Release Notes*.
- Install and verify the operation of the BEA Tuxedo product. Refer to the "Special Instructions for Installing the Tuxedo Core" section of this guide for installation tips and the *BEA Tuxedo Installation Guide* at http://edocs.beasys.com/tuxedo/tux65/index.htm for more information.

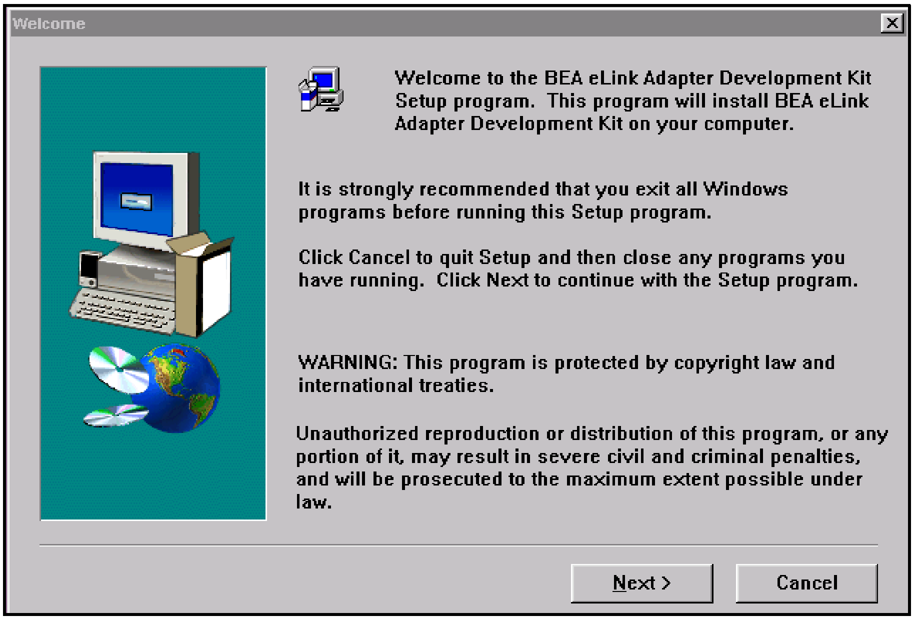
Installing on the Windows NT Platform

The eLink Adapter Development Kit software is available only for Version 4.0 of the Windows NT platform.

Perform the following steps to install the eLink ADK software on a Windows NT system:

1. Insert the product CD-ROM and click the **Run** option from the **Start menu**. The **Run** window displays. Click the **Browse** button to select the CD-ROM drive. Select the `winnt` directory and select the `Setup.exe` program. Click **OK** to run the executable and begin the installation. The **Welcome** window displays as shown in Figure 4-1. Click **Next** to continue with the installation.

Figure 4-1 Welcome Window



Welcome to the BEA eLink Adapter Development Kit Setup program. This program will install BEA eLink Adapter Development Kit on your computer.

It is strongly recommended that you exit all Windows programs before running this Setup program.

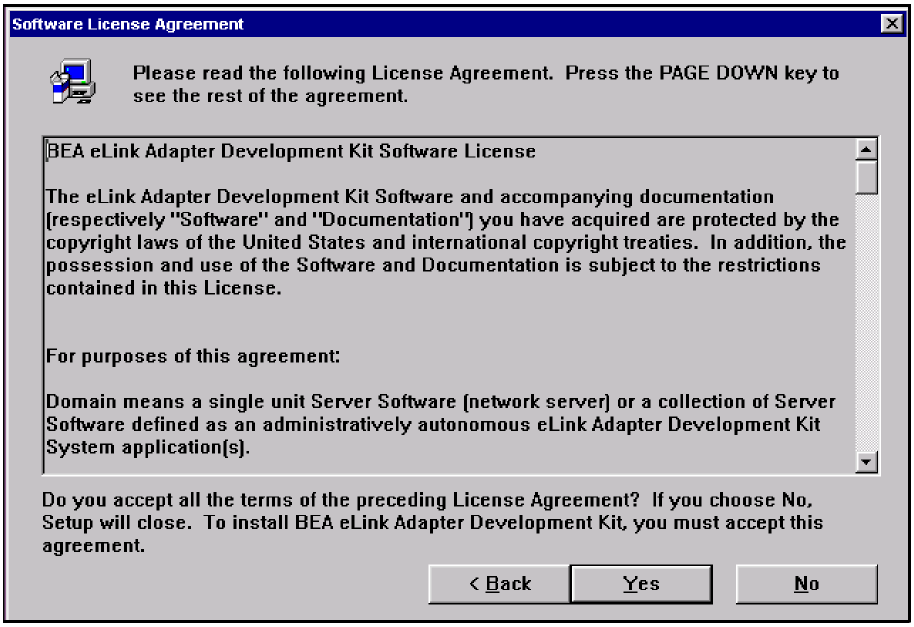
Click Cancel to quit Setup and then close any programs you have running. Click Next to continue with the Setup program.

WARNING: This program is protected by copyright law and international treaties.

Unauthorized reproduction or distribution of this program, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under law.

2. The **License Agreement** window displays as shown in Figure 4-2. Read the license agreement information, and click **Yes** to continue with the installation.

Figure 4-2 License Agreement Window



Software License Agreement

Please read the following License Agreement. Press the PAGE DOWN key to see the rest of the agreement.

BEA eLink Adapter Development Kit Software License

The eLink Adapter Development Kit Software and accompanying documentation (respectively "Software" and "Documentation") you have acquired are protected by the copyright laws of the United States and international copyright treaties. In addition, the possession and use of the Software and Documentation is subject to the restrictions contained in this License.

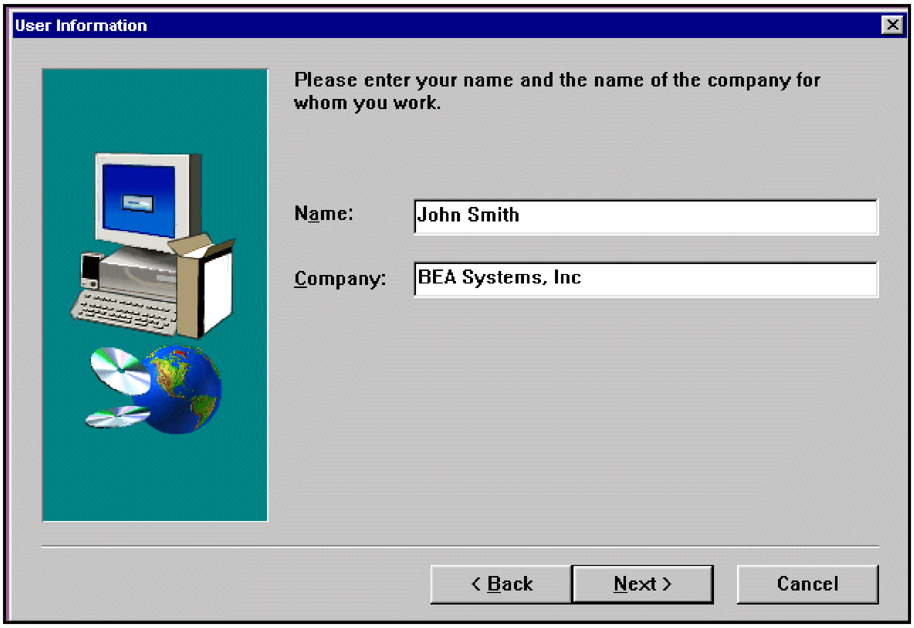
For purposes of this agreement:

Domain means a single unit Server Software (network server) or a collection of Server Software defined as an administratively autonomous eLink Adapter Development Kit System application(s).

Do you accept all the terms of the preceding License Agreement? If you choose No, Setup will close. To install BEA eLink Adapter Development Kit, you must accept this agreement.

3. The **User Information** window displays as shown in Figure 4-3. Enter your name in the **Name** field. Enter the name of your company in the **Company** field. Click **Next** to continue with the installation.

Figure 4-3 User Information Window



User Information

Please enter your name and the name of the company for whom you work.

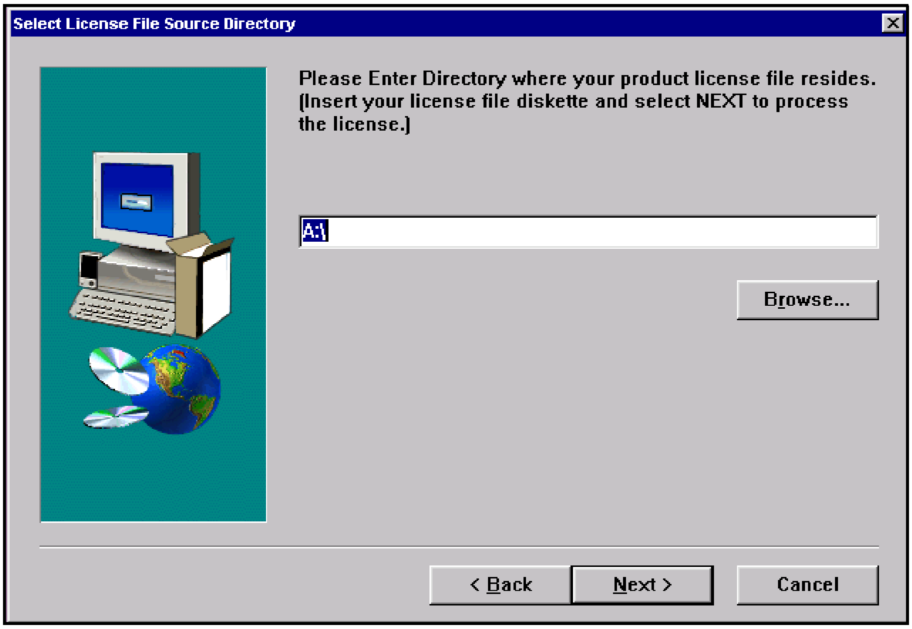
Name: John Smith

Company: BEA Systems, Inc

4. After you click **Next**, the **Select License File Source Directory** window displays as shown in Figure 4-4.

   a. Enter the Directory and Path or click the **Browse** button to display the **Choose Folder** pop-up window as shown in Figure 4-5

Figure 4-4 Select License File Source Directory Window

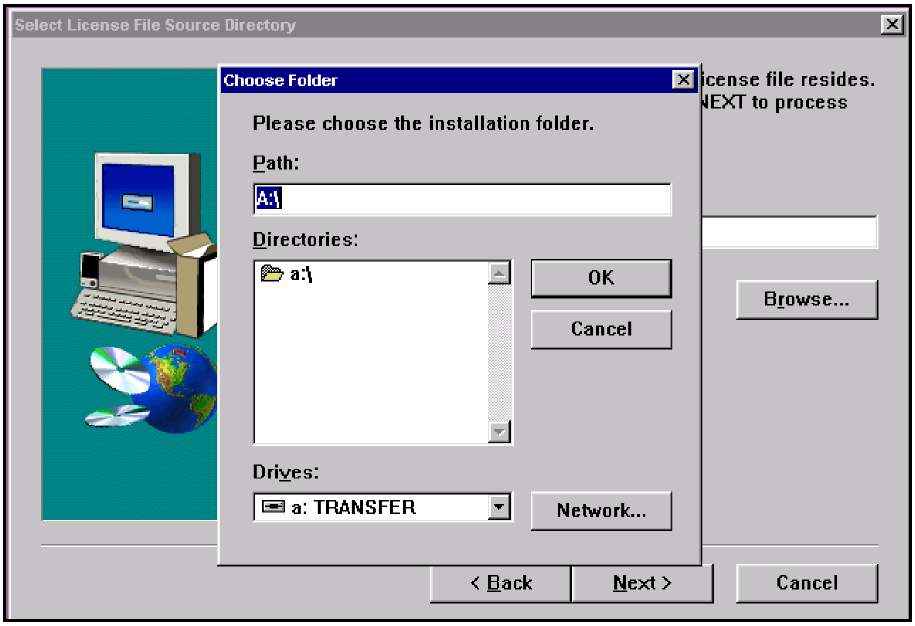


Please Enter Directory where your product license file resides. (Insert your license file diskette and select NEXT to process the license.)

A:\

b. If you clicked **Browse**, locate the License file and click **OK** to return to the **Select License File Source Directory** window**.** Then click **Next** to continue with the installation process.

Figure 4-5 Choose Folder Pop-up Window



Please choose the installation folder.

Path:

A:\

Directories:

a:\

Drives:

a: TRANSFER

5. **If,**

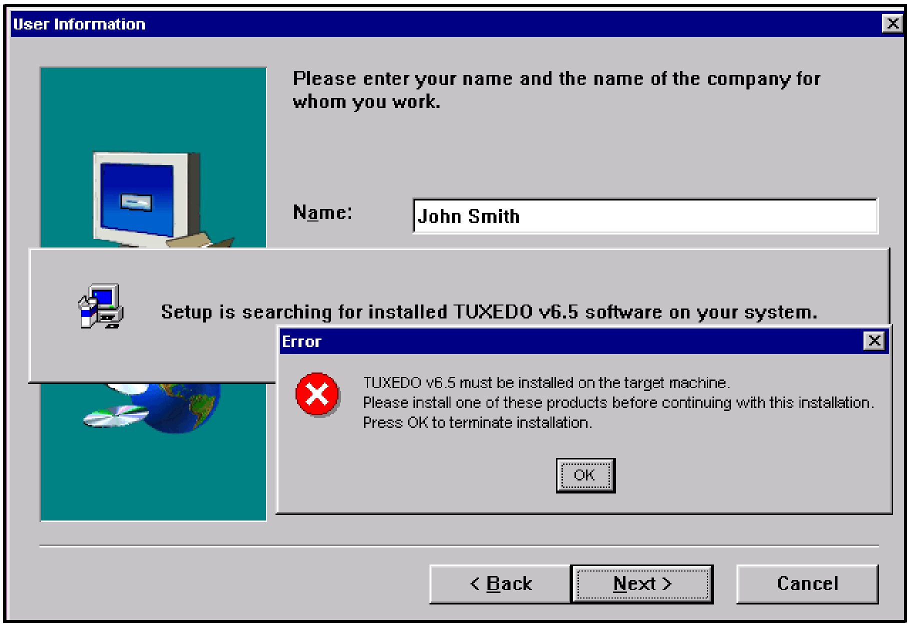
   Tuxedo is already installed on your system,
   The installation begins and a progress bar displays the status. The eLink ADK components install into the Tuxedo directory. You may abort the installation process anytime prior to completion by clicking the **Cancel** button.

   When the installation completes, the **Setup Complete** window shown in Figure 4-7 notifies you that the eLink ADK software is installed on your system.

Warning: If Windows NT is your execution environment, BEA TUXEDO should be installed first and the eLink ADK should be installed within the same directory. If you install the eLink ADK outside of the Tuxedo directory, you will need to copy the files into the Tuxedo directory for processing of data mapping service requests.Click **Yes** to continue the installation or **No** to quit.

   Tuxedo is NOT already installed on your system,
   The **Error** pop-up window displays as shown in Figure 4-6. Click **OK** on the pop-up window to terminate the installation process. Install Tuxedo 6.5 on your system (see warning above). Reinitiate the installation process starting with step one of these installation instructions.

Figure 4-6 Tuxedo 6.5 Installation Error Pop-Up Window



User Information

Please enter your name and the name of the company for whom you work.

Name: John Smith

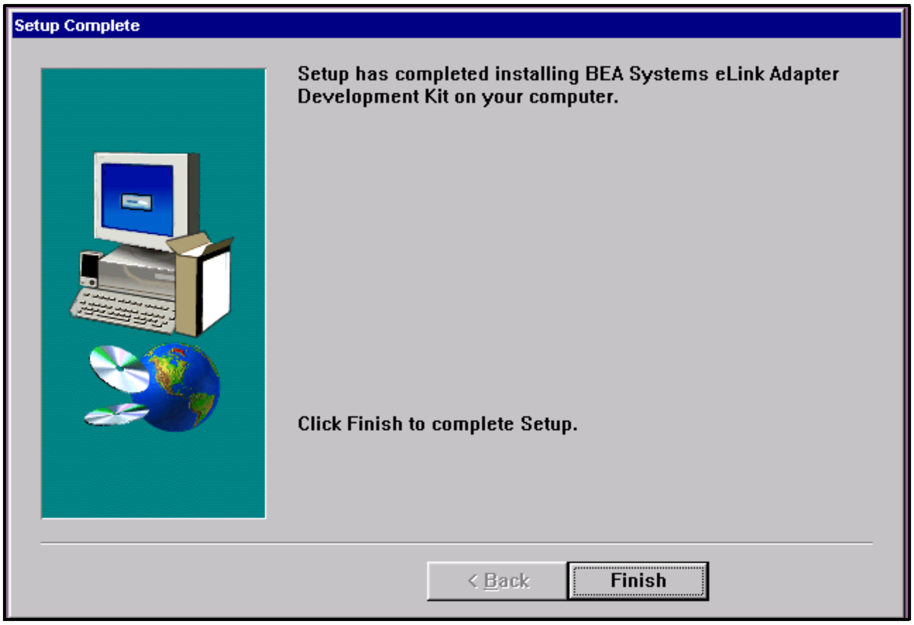
Setup is searching for installed TUXEDO v6.5 software on your system.

Error

TUXEDO v6.5 must be installed on the target machine.
Please install one of these products before continuing with this installation.
Press OK to terminate installation.

6. The **Setup Complete** window notifies you that the eLink ADK software is installed on your system. Click **Finish** to complete the setup process.

Figure 4-7 Setup Complete



Setup has completed installing BEA Systems eLink Adapter Development Kit on your computer.

Click Finish to complete Setup.

Installing on the HP-UX, AIX, Solaris, and Compaq TRU64 UNIX Platforms

This section explains how to install the eLink Adapter Development Kit software on the following execution platforms.

- HP-UX 10.20 or 11.00
- AIX v4.3.x
- SUN Solaris 2.6 or 7
- Compaq Tru64 UNIX 5.0

Warning: You must install the eLink ADK execution components within the eLink Platform directory.

To install the eLink ADK software, you run the `install.sh` script. This script installs all the necessary software components.

Perform the following steps to install the eLink ADK software on a supported Unix platform:

1. Log on as root to install the eLink ADK software.

```
$ su -
Password:
```

2. Access the CD-ROM device.

```
# ls -1 /dev/cdrom

total 0

brw-rw-rw-     1 root     sys     22,   0 January 5 10:55 c1b0t010
```

3. Mount the CD-ROM.

```
# mount -r -F cdfs /dev/cdrom/c1b0t0l0 /mnt
```

4. Change the directory to your CD-ROM device.

```
# cd /mnt
```

5. List the CD-ROM contents.

```
# ls
```

```
install.sh  hp
```

6. Execute the installation script.

```
# sh ./install.sh
```

7. The installation script runs and prompts you for responses.

Listing 4-1 Install.sh Script Prompts

```
cmadm@dalhpw1:/cmhome/dist/balkan-1> ls
alpha/      hp/        ibm/       install.sh*  sun5x/      winnt/
cmadm@dalhpw1:/cmhome/dist/balkan-1> sh install.sh

01) alpha/tru64      02) hp/hpux1020      03) hp/hpux11
04) ibm/aix43        05) sun5x/sol26      06) sun5x/sol7


Install which platform’s files? [01-6, q to quit, l for list]: 2

** You have chosen to install from hp/hpux1020 **

BEA eLink Adapter Development Kit Release 1.1

This directory contains the BEA eLink Adapter Development Kit System
for
HP-UX 10.20 on 9000/800 series.


Is this correct? [y,n,q]: y

To terminate the installation at any time
press the interrupt key,
typically <del>, <break>, or <ctrl+c>.

The following packages are available:

  1     adk             BEA eLink Adapter Development Kit

Select the package(s) you wish to install (or ’all’ to install
all packages) (default: all) [?,??,q]:

BEA eLink Adapter Development Kit
(9000) Release 1.1
Copyright (c) 2000 BEA Systems, Inc.
```

```
All Rights Reserved.
Distributed under license by BEA Systems, Inc.
BEA eLink is a trademark of BEA Systems, Inc.


Directory where ADK Adapter files are to be installed
(Enter your Tuxedo directory path) [?,q]: /work/cmadm/tux65

Using /work/cmadm/tux65 as the ADK Adapter base directory

Determining if sufficient space is available ...
528 blocks are required
3783706 blocks are available to /work/cmadm/tux65

Unloading /cmhome/dist/balkan-1/hp/hpux1020/adk/ADKT65.Z ...
adkdemo/adkdemo.text
adkdemo/email/README.DEMO
adkdemo/email/adkdemo.text
adkdemo/email/config.c
adkdemo/email/config.h
adkdemo/email/email.cfg
adkdemo/email/emailclient.c
adkdemo/email/emaildemo.mak
adkdemo/email/emaildemox.ubb
adkdemo/email/emailmessage.txt
adkdemo/email/emailserver.c
adkdemo/email/emailserver.cfg
adkdemo/email/emailserver.h
adkdemo/email/emailtools.h
adkdemo/email/makeclient.scr
adkdemo/email/makeserver.scr
adkdemo/email/setadk.scr
adkdemo/inbound/elinkdi.mak
adkdemo/inbound/inbound.c
adkdemo/inbound/input.txt
adkdemo/inbound/readme
adkdemo/inbound/setenv.sh
adkdemo/inbound/simpserv.c
adkdemo/inbound/table.fml
adkdemo/inbound/ubb.inbound
adkdemo/outbound/adkdemo.c
adkdemo/outbound/adkdemo.cfg
adkdemo/outbound/adkdemo.h
adkdemo/outbound/adkdemox.ubb
adkdemo/outbound/config.c
adkdemo/outbound/config.h
adkdemo/outbound/democlient.c
adkdemo/outbound/elinkdo.mak
adkdemo/outbound/foofns.c
```

```
adkdemo/outbound/foofns.h
adkdemo/outbound/makeclient.scr
adkdemo/outbound/makeserver.scr
adkdemo/outbound/readme.demo
adkdemo/outbound/setadk.scr
bin/lic.sh
include/adkfns.h
include/adklog.h
include/adktypes.h
include/fmlfns.h
lib/libadk.sl.1.10
lib/libcomm.sl
lib/libsmtp.sl
490 blocks
... finished

Changing file permissions...
... finished

If your license file is accessible, you may install it now.
Install license file? [y/n]: n

Please don't forget to use lic.sh located in your product bin
directory
to install the license file from the enclosed floppy.
Refer to your product Release Notes for details on how to do this.


Installation of BEA eLink Adapter Development Kit was successful

Please don't forget to fill out and send in your registration card
cmadm@dalhpw1:/cmhome/dist/balkan-1>
```

CHAPTER

5 Configuring and Running the Sample Adapters

The eLink Adapter Development Kit includes a Sample Application to eLink Adapter, a Sample eLink to Application Adapter, and a Sample E-mail Adapter.

This chapter discusses the following topics:

- Demo Prerequisites for UNIX
- The Sample Application to eLink Adapter
- The Sample eLink to Application Adapter
- The Sample E-Mail Adapter

Demo Prerequisites for UNIX

If you run the demos on a UNIX operating system, there are several prerequisites:

- Before building the demo, you **must** have GNU Make Version 3.74 or higher. If you do not have Make, you can download it from http\\www.gnu.org. Select the **`Software`** item and then scroll down to **`make`**.
- Check the machine entry in your `elinkdo.mak` and `elinkdi.mak` files by performing the following procedure:

- Open the `.mak` file in a text editor.
- Find the following line(s), which may be entered multiple times for multiple machines:
  `ifeq ADK_MACHINE #`, where `#` equals a machine ID per the following table.

Table 5-1 UNIX Machine ID Numbers

Operating System	ID #
AIX 4.3	49
Compaq Tru64 UNIX 5.0	37
HP-UX 10.20	36
HP-UX 11.0	40
Solaris	35

- Verify that the information under the machine ID line is correct for the identified operating system. If it is not correct, change the information to reflect your system configuration.
- Save the edited `.mak` file.

The Sample Application to eLink Adapter

The Sample Application to eLink Adapter provides an example of the general form of an application to eLink adapter. In particular, it provides an example of using a Tuxedo daemon service to poll an external source for requests. In the case of the Sample Application to eLink Adapter, the external source is just a text file that contains request strings. The Sample Application to eLink Adapter also provides a complete example of the use of **FML32**. At this time, the Sample Application to eLink Adapter does not provide a complete code example of an adapter because it does not read information from a configuration file, it does not use the standard `eLA_log()` function provided for logging message by adapters, and it does not check a license file. For a more complete code example of an adapter, see the Sample eLink to Application Adapter.

Before installing and running the Sample Application to eLink Adapter, you must install **Tuxedo**.

Configuring the Sample Application to eLink Adapter

To configure the Sample Application to eLink Adapter:

1. Edit `SETENV.BAT` for NT, or `setenv.sh` for Unix to reflect your environment.

Note: For NT, Tuxedo sets `TUXDIR` and adds `TUXDIR\bin` to the `PATH` environment variable. For Unix, Tuxedo generates a `TUX.ENV` file that sets these variables for you.

You need to set the `INCLUDE, LIB, APPDIR,` TUXCONFIG, `FIELDTBLS32`, and `FLDTBLDIR32` variables on both NT and Unix, and the `SHLIB_PATH` (or `LD_LIBRARY_PATH` or `LIBPATH`) variable on Unix.

Make sure that each of the following variables is set correctly:

- `TUXDIR=<Base directory of the Tuxedo software>`
- `APPDIR=<Base directory of the sample application>`
- `PATH must include $TUXDIR/bin`
- `TUXCONFIG=<Full pathname of the binary tuxconfig file>`
- `FLDTBLDIR32=<Base directory of the sample application>`
- `FIELDTBLS32=table.fml (FML32 table for this application)`
- `LD_LIBRARY_PATH must include $TUXDIR/lib on systems that use shared libraries (except HP-UX and AIX).`
- `SHLIB_PATH (HP-UX only) must include $TUXDIR/lib`
- `LIBPATH (AIX only) must include $TUXDIR/lib`
- `ADK_MACHINE must be set to the appropriate machine ID # as referenced in Table 5-1`

2. Execute `SETENV.BAT` for NT, or `setenv.sh` for Unix to complete the installation.

Building and Running the Sample Application to eLink Adapter

At this point, you are ready to build and run the Sample Application to eLink Adapter included in ADKDEMO\INBOUND.

1. Because the sample inbound adapter runs as a Tuxedo server and the imitation third-party “software” is just a file that is used for input, there is no Tuxedo client to build, but there are two **servers** that must be built.

   To build the servers, execute:

   ```
   make -felinkdi.mak
   ```

   or use the commands:

   ```
   buildserver -o simpserv -f simpserv.c -s TOUPPER
   buildserver -o inbound -f inbound.c -s DAEMON
   ```

Note: For NT, the default compiler is the MS VC++ command line compiler, CL.EXE. While the compiler can be changed by setting the CC environment variable, there is no guarantee that the .LIB files supplied with either Tuxedo or the ADK will work with any other compiler.

2. Edit the `ubb.inbound` file. The `APPDIR, TUXDIR` and `TUXCONFIG` variables need to be set independently of external variables. Be sure that the variables set in the `ubb.inbound` file match your environment variables. The machine ID field must also be set. The machine ID field on Windows NT (in this instance DALNT10) should be entered in upper case.

3. Execute `tmloadcf` with `ubb.inbound` as an argument. This converts the `ubb.inbound` text file into a binary **`TUXCONFIG`** file.

   ```
   tmloadcf -y ubb.inbound
   ```

4. Execute `tmboot` to load the `ADKDEMO` inbound application.

   ```
   tmboot -y
   ```

   The demo inbound server is booted and runs until it has processed all requests in the file `input.txt`. The outputs are written to `stdout`.

5. Execute `tmshutdown` to shut down the `ADKDEMO` inbound application.

The Sample eLink to Application Adapter

Before installing and running the Sample eLink to Application Adapter, you must install **Tuxedo** and the ADK.

Configuring the Sample eLink to Application Adapter

To configure the Sample Application to eLink Adapter:

1. Edit `SETADK.BAT` for NT or `SETADK.SCR` for Unix to reflect your environment.

Note: For NT, Tuxedo sets `TUXDIR` and adds `TUXDIR\bin` to the `PATH` environment variable. For Unix, Tuxedo generates a `TUX.ENV` file that sets these variables for you. You need to set the `INCLUDE`, `LIB, APPDIR` and `TUXCONFIG` variables on both NT and Unix, and the `SHLIB_PATH` variable on Unix.

2. Execute `SETADK.BAT` for NT or `SETADK.SCR` for Unix to complete the installation.

Building and Running the Sample eLink to Application Adapter

At this point, you are ready to build and run the sample eLink to application adapter included in `ADKDEMO`.

1. Build the client and server as appropriate for your operating system:
   - For NT, execute MAKECLIENT.BAT and MAKESERVER.BAT.
   - For Unix, enter make -felinkdo.mak or execute MAKECLIENT.SCR and MAKESERVER.SCR.

   These batch or script files invoke **`BUILDCLIENT`** and **`BUILDSERVER`** to build `DEMOCLIENT.EXE` and `ADKDEMO.EXE`. No changes should be required as long as **`BUILDSERVER`**, **`BUILDCLIENT`** and your C compiler can be found.

Note: For NT, the default compiler is the MS VC++ command line compiler, CL.EXE. While this compiler can be changed by setting the CC environment variable, there is no guarantee that the .LIB files supplied with either **Tuxedo** or the ADK will work with any other compiler.

2. Edit the `ADKDEMO.UBB` file for NT or `ADKDEMOX.UBB` file for Unix. The `APPDIR`, `TUXDIR` and **`TUXCONFIG`** variables need to be set independently of external variables. The machine ID field on NT (in this instance DALNT10) should be entered in upper case.

Note: The `ADKDEMO.CFG` file referenced by both `UBB` files need not be edited, because it is installation independent.

3. Execute `tmloadcf` with the appropriate `UBB` file as an argument to convert the `UBB` text file into a binary **`TUXCONFIG`** file.
4. Execute `tmboot` to load the `ADKDEMO` application.
5. Run `DEMOCLIENT` with a variety of arguments to exercise the **server**.
6. Execute `tmshutdown` to shut down the `ADKDEMO` application.

The Sample E-Mail Adapter

The Sample E-Mail Adapter is an eLink to application adapter that consists of a Tuxedo client and server. The E-Mail Adapter’s SEND service allows a user to connect to an SMTP Server to send an E-Mail. The RECEIVE service is not implemented so you must add the functionality to connect to a POP3 server to receive E-Mail.

The E-Mail client uses the contents of the `email.cfg` file to send information to the E-Mail server. The email.cfg file contains the following information in the form of `field name = field contents`. Edit the .cfg file with information appropriate for your system.

```
server=your.SMTPServer.com
domain=your.SMTPDomain.com
sender=sender@email.com
```

```
recipient=recipient@email.com
subject=message subject
file=emailmessage.txt
```

When the E-Mail client is invoked, it reads the E-Mail information from the email.cfg file, puts the .cfg file fields into an FML32 buffer, and then sends that buffer to the E-Mail server. When the server receives the E-Mail buffer, it processes the buffer and uses emailtools to configure the message and send it to the specified SMTP server. When the server receives confirmation that the message was sent, the server returns the FML32 buffer back to the E-Mail client and the process terminates.

Invoking the Sample E-Mail Adapter

Invoke the Sample E-Mail Adapter with a command line in the following format:

```
$ emailclient <service> [ -f email.cfg ])
```

For example, to invoke the Sample E-Mail Adapter using default information contained in the `email.cfg` file, use the following entry for UNIX:

```
$ emailclient send -f email.cfg
```

For Windows NT, enter the command at the appropriate directory prompt:

```
C:\emailclient send -f email.cfg
```

Or, to invoke the Sample E-Mail Adapter using your own information rather than using parameters specified in the `email.cfg` file, enter the emailclient command without specifying the .cfg file as the following UNIX example shows:

```
$ emailclient send
```

After submitting the command, you will be prompted to enter the following parameters, where the text in brackets equals the parameter values that you wish to use:

```
Enter Enter SMTP server (your.SMTPServer.com)=[your SMTP server]
Enter SMTP domain (your.SMTPDomain.com)=[your SMTP domain]
Enter Recipient of email (recipient@email.com)=[your recipient’s E-Mail address]
Enter Sender's email (sender@email.com)=[your sender’s E-Mail address]
Enter Subject of email=[your message subject]
Enter file loacation of message=[your emailmessage.txt]
```

Configuring the Sample E-Mail Adapter

To configure the Sample E-Mail Adapter:

1. Edit `SETADK.BAT` for NT or `SETADK.SCR` for Unix to reflect your environment.

Note: For NT, Tuxedo sets `TUXDIR` and adds `TUXDIR\bin` to the `PATH` environment variable. For Unix, Tuxedo generates a `TUX.ENV` file that sets these variables for you. You need to set the `INCLUDE, LIB, APPDIR, ADKDIR, ADK_MACHINE`, and `TUXCONFIG` variables on both NT and UNIX, and the `SHLIB_PATH` on HP-UX, `LIBPATH` on AIX, and `LD_LIBRARY_PATH` on all other UNIX platforms.

2. Execute `SETADK.BAT` for NT or `SETADK.SCR` for Unix to complete the installation.

Building and Running the Sample E-Mail Adapter

At this point, you are ready to build and run the sample E-Mail Adapter.

1. Build the client and server as appropriate for your operating system:
   - For NT, execute MAKECLIENT.BAT and MAKESERVER.BAT.
   - For Unix, enter make -f emaildemo.mak or execute MAKECLIENT.SCR and MAKESERVER.SCR.

   Note: The GNU Make Utility is used for the make function. Refer to Demo Prerequisites for UNIX for more information about GNU Make.

   These batch or script files invoke **`BUILDCLIENT`** and **`BUILDSERVER`** to build `EMAILCLIENT.EXE` and `EMAILSERVER.EXE`. No changes should be required as long as **`BUILDSERVER`**, **`BUILDCLIENT`** and your C compiler can be found.

Note: For NT, the default compiler is the MS VC++ command line compiler, CL.EXE. While this compiler can be changed by setting the CC environment variable, there is no guarantee that the .LIB files supplied with either **Tuxedo** or the ADK will work with any other compiler.

2. Edit the `EMAILDEMO.UBB` file for NT or `EMAILDEMOX.UBB` file for Unix. The `APPDIR, TUXDIR` and **`TUXCONFIG`** variables need to be set independently of external variables. The machine ID field on NT (in this instance DALNT10) should be entered in upper case.

Note: The `EMAILSERVER.CFG` file referenced by both `UBB` files need not be edited, because it is installation independent.

3. Execute `tmloadcf` with the appropriate `UBB` file as an argument to convert the `UBB` text file into a binary **`TUXCONFIG`** file.
4. Execute `tmboot` to load the `EMAILSERVER` application.
5. Run `EMAILCLIENT` with a variety of arguments to exercise the **server**.
6. Execute `tmshutdown` to shut down the `EMAILSERVER` application.

A eLink Adapter Development Kit References

This section discusses the following topics:

- Configuration Processing API
- Hash Table API
- Utility Functions and Macros
- Definitions and Typedefs

Configuration Processing API

This section provides detailed descriptions of the configuration processing API used by the ADK listed in the order that the functions will most likely be used. Following is an alphabetical list of the configuration processing API functions and a reference to where details about the function can be found in this appendix.

Table A-1 Alphabetical Cross-Reference List of Configuration Processing API Functions

Refer to
eLA_CloseTagFile
eLA_CloseTagHandle
eLA_DestHashTable
eLA_get
eLA_GetFieldMap
eLA_GetFirstField
eLA_GetFirstProperty
eLA_GetFirstSection
eLA_GetNextField
eLA_GetNextProperty
eLA_GetNextSection
eLA_GetPropertyValue
eLA_hash
eLA_InitHashTable
eLA_OpenTagFile
eLA_put

eLA_OpenTagFile

`eLA_OpenTagFile` opens a Tag file, reads the data into memory. Some preliminary processing is performed, such as deleting leading and trailing white space, removing blank lines, converting section names and tag data to uppercase and indexing the data. Section names are always preceded by an asterisk (*). Tag data specifically refers to a string that is to the left of an equal sign on a line. If a string is not preceded by an asterisk or to the left of an equal sign on a line, then it will not be uppercased. Finally, the file is closed and a handle to the data is returned. This handle is used by the `eLA_GetFirstSection` function and should be closed using `eLA_CloseTagFile` when no longer needed. Multiline entries are allowed. If the last non white space character on any line is a \, the next line will be concatenated to the current line. A blank line will terminate the multiline entry.

Prototype

```
ADK_CFG_HANDLE eLA_OpenTagFile(char * TagFileName)
```

where

`TagFileName` is the name of the Tag (or .INI) file.

Return

Return	**Description**
`ADK_INVALID_HANDLE_VALUE`	On any error, such as invalid file name, malloc problems, etc.
Anything else	Success

Example

```
#include “adkfns.h”
...
ADK_CFG_HANDLE fHandle;
char FileName[256];
...
strcpy(FileName, “MYSAMPLE.INI”);
fHandle = eLA_OpenTagFile(FileName);
if(fHandle == ADK_INVALID_HANDLE_VALUE)
  printf(“Unable to process file %s\n”,FileName);
else
  printf(“File %s opened\n”, FileName);
...
eLA_CloseTagFile(fHandle);
```

eLA_CloseTagFile

`eLA_CloseTagFile` closes a handle returned by `eLA_OpenTagFile` and frees any internally allocated resources. It can be called with an invalid handle.

Prototype

```
int eLA_CloseTagFile(ADK_CFG_HANDLE TagFileHandle)
```

where

`TagFileHandle` is the handle returned by `eLA_OpenTagFile`.

Return

Return	Description
`ADK_SUCCESS`	Success
Anything else	Error

Example

```
#include “adkfns.h”
...
ADK_CFG_HANDLE fHandle;
...
fHandle = eLA_OpenTagFile(FileName);
...
eLA_CloseTagFile(fHandle);
```

eLA_CloseTagHandle

`eLA_CloseTagHandle` closes a handle returned by `eLA_GetFirstSection` and frees any internally allocated resources. It also can be called with an invalid handle.

Prototype `int eLA_CloseTagHandle(ADK_CFG_HANDLE GenericHandle)`

where

`GenericHandle` is the handle returned by `eLA_GetFirstSection`.

Return

Return	Description
`ADK_SUCCESS`	Success.
Anything else	Error.

Example

```
#include “adkfns.h”
...
ADK_CFG_HANDLE fHandle, sHandle;
 ...
fHandle = eLA_OpenTagFile(FileName);
```

```
...
if(fHandle != ADK_INVALID_HANDLE_VALUE)
  sHandle = eLA_GetFirstSection(fHandle, "SERVICE");
...
eLA_CloseTagFile(fHandle);
eLA_CloseTagHandle(sHandle);
```

eLA_GetFirstSection

`eLA_GetFirstSection` searches the INI file memory image for the desired sections (all text matching done after conversion to upper case). These locations are indexed, and a handle to this data is returned. This handle is used by the `eLA_GetNextSection`, `eLA_GetFirstProperty`, `eLA_GetNextProperty` and `eLA_GetPropertyValue`. It should be freed by `eLA_CloseTagHandle` when no longer needed.

Prototype

```
ADK_CFG_HANDLE eLA_GetFirstSection(ADK_CFG_HANDLE TagFileHandle,
      char * SectionName)
```

where

`TagFileHandle` is the handle returned by `eLA_OpenTagFile`.

`SectionName` is the name of the section desired, excluding “*” (SectionName is provided by the API). The text is converted to upper case before searching.

Return

Return	**Description**
`ADK_INVALID_HANDLE_VALUE`	On any error, such as invalid handle in argument, malloc problems, or not finding section name.
Anything else	Success

Example

```
#include “adkfns.h”
...
ADK_CFG_HANDLE fHandle, sHandle;
...
fHandle = eLA_OpenTagFile(FileName);
...
if(fHandle != ADK_INVALID_HANDLE_VALUE)
  sHandle = eLA_GetFirstSection(fHandle, "SERVICE");
...
```

```
eLA_CloseTagFile(fHandle);
eLA_CloseTagHandle(sHandle);
```

eLA_GetNextSection

`eLA_GetNextSection` updates the information in the section handle returned by `eLA_GetFirstSection` and point to the next occurrence of the section in question.

Prototype `int eLA_GetNextSection(ADK_CFG_HANDLE SectionHandle)`

where

`SectionHandle` is the handle returned by `eLA_GetFirstSection`.

Return

Return	Description
ADK_INVALID_HANDLE_VALUE	Invalid handle in argument.
ADK_ERROR_LAST_SECTION	No more sections.
ADK_SUCCESS	Success

Example

```
#include “adkfns.h”
...
ADK_CFG_HANDLE fHandle, sHandle;
int rc;
...
fHandle = eLA_OpenTagFile(FileName);
...
if(fHandle != ADK_INVALID_HANDLE_VALUE)
  sHandle = eLA_GetFirstSection(fHandle, "SERVICE");
  if(sHandle != ADK_INVALID_HANDLE_VALUE)
    { rc = ADK_SUCCESS;
      while(rc == ADK_SUCCESS);
      { rc = eLA_GetNextSection(sHandle);
        ...
      }
    }
...
eLA_CloseTagFile(fHandle);
eLA_CloseTagHandle(sHandle);
```

eLA_GetFirstProperty

`eLA_GetFirstProperty` retrieves the Tag/Value pair for the first property for a particular section.

Prototype

```
int eLA_GetFirstProperty(ADK_CFG_HANDLE SectionHandle,
                          char * TagBuffer, size_t TagLength,
                          char * ValueBuffer, size_t ValueLength);
```

where

`SectionHandle` is the handle returned by `eLA_GetFirstSection`.

`TagBuffer` is address of the tag data return buffer.

`TagLength` is the size of the tag data buffer (including NULL).

`ValueBuffer` is the address of the value data return buffer.

`ValueLength` is the size of the value data buffer (including NULL).

Return

Return	**Description**
`ADK_INVALID_HANDLE_VALUE`	Invalid handle in argument.
`ADK_ERROR_NO_PROPERTIES`	No property lines for section.
`ADK_ERROR_NO_VALUE`	No text following the = in the tag line.
`ADK_ERROR_NO_TAG`	No text preceding the = in the tag line.
`ADK_ERROR_BUFFER_OVERFLOW`	Not enough room for either tag or value data.

Example

```
#include “adkfns.h”
...
ADK_CFG_HANDLE fHandle, sHandle;
char tBuffer[100], vBuffer[100];
int rc, pc;
...
fHandle = eLA_OpenTagFile(FileName);
...
if(fHandle != ADK_INVALID_HANDLE_VALUE)
  sHandle = eLA_GetFirstSection(fHandle, "SERVICE");
```

```
     if(sHandle != ADK_INVALID_HANDLE_VALUE)
       { rc = ADK_SUCCESS;
         while(rc == ADK_SUCCESS);
         { pc = eLA_GetFirstProperty(sHandle, tBuffer,
              sizeof(tBuffer),vBuffer,sizeof(vBuffer);
           ...
           rc = eLA_GetNextSection(sHandle);
           ...
         }
       }
...
eLA_CloseTagFile(fHandle);
eLA_CloseTagHandle(sHandle);
```

eLA_GetNextProperty

`eLA_GetNextProperty` retrieves the Tag/Value pair for the second and successive properties for a particular section.

Prototype

```
int eLA_GetNextProperty(ADK_CFG_HANDLE SectionHandle,
                          char * TagBuffer, size_t TagLength,
                          char * ValueBuffer, size_t ValueLength);
```

where

`SectionHandle` is the handle returned by `eLA_GetFirstSection`.

`TagBuffer` is the address of the tag data return buffer.

`TagLength` is the size of the tag data buffer (including NULL).

`ValueBuffer` is the address of the value data return buffer.

`ValueLength` is the size of the value data buffer (including NULL).

Return

Return	**Description**
`ADK_INVALID_HANDLE_VALUE`	Invalid handle in argument.
`ADK_ERROR_LAST_PROPERTY`	No more properties for this section.
`ADK_ERROR_NO_VALUE`	No text following the = in the tag line.

Return	Description
ADK_ERROR_NO_TAG	No text preceding the = in the tag line.
ADK_ERROR_BUFFER_OVERFLOW	Not enough room for either tag or value data.

Example

```
#include “adkfns.h”
...
ADK_CFG_HANDLE fHandle, sHandle;
char tBuffer[100], vBuffer[100];
int rc, pc;
...
fHandle = eLA_OpenTagFile(FileName);
...
if(fHandle != ADK_INVALID_HANDLE_VALUE)
  sHandle = eLA_GetFirstSection(fHandle, "SERVICE");
  if(sHandle != ADK_INVALID_HANDLE_VALUE)
    { rc = ADK_SUCCESS;
      while(rc == ADK_SUCCESS);
      { pc = eLA_GetFirstProperty(sHandle, tBuffer,
            sizeof(tBuffer),vBuffer, sizeof(vBuffer);
        ...
        pc = eLA_GetNextProperty(sHandle, tBuffer,
            sizeof(tBuffer),vBuffer, sizeof(vBuffer);
        ...
        rc = eLA_GetNextSection(sHandle);
        ...
      }
   }
...
eLA_CloseTagFile(fHandle);
eLA_CloseTagHandle(sHandle);
```

eLA_GetPropertyValue

eLA_GetPropertyValue retrieves the Value data for the first occurrence of a particular tag in a given section. Text fields are converted to upper case prior to searching.

Prototype

```
int eLA_GetPropertyValue(ADK_CFG_HANDLE SectionHandle,
     char * TagName,char * ValueBuffer, size_t ValueLength);
```

where

`SectionHandle` is the handle returned by `eLA_GetFirstSection`.

`TagName` is the tag to search for.

`ValueBuffer` is the address of the value data return buffer.

`ValueLength` is the size of the value data buffer (including NULL).

Return

Return	Description
`ADK_INVALID_HANDLE_VALUE`	Invalid handle in argument.
`ADK_ERROR_PROPERTY_NOT_FOUND`	No tag matching input found.
`ADK_ERROR_NO_VALUE`	No text following the = in the tag line.
`ADK_ERROR_NO_TAG`	No text preceding the = in the tag line.
`ADK_ERROR_BUFFER_OVERFLOW`	Not enough room for value data.

Example

```
#include “adkfns.h”
...
ADK_CFG_HANDLE fHandle, sHandle;
char tBuffer[100], vBuffer[100];
int rc, pc;
...
fHandle = eLA_OpenTagFile(FileName);
...
if(fHandle != ADK_INVALID_HANDLE_VALUE)
  sHandle = eLA_GetFirstSection(fHandle, "SERVICE");
  if(sHandle != ADK_INVALID_HANDLE_VALUE)
    { rc = ADK_SUCCESS;
      while(rc == ADK_SUCCESS);
      { pc = eLA_GetPropertyValue(sHandle, "NAME" vBuffer,
               sizeof(vBuffer);
        ...
        rc = eLA_GetNextSection(sHandle);
        ...
      }
    }
...
eLA_CloseTagFile(fHandle);
eLA_CloseTagHandle(sHandle);
```

eLA_GetFieldMap

`eLA_GetFieldMap` searches the adapter specific configuration file memory image for the named *FIELDMAP section. A handle to the data is returned, which is then used by `eLA_GetFirstField` and `eLA_GetNextField` to retrieve each line in order. This handle should be freed by `eLA_CloseTagHandle` when no longer needed. (This function is somewhat analogous to `eLA_GetFirstSection`).

Prototype

```
ADK_CFG_HANDLE eLA_GetFieldMap(ADK_CFG_HANDLE TagFileHandle
        char * FieldMapName)
```

where

`TagFileHandle` is the handle returned by `eLA_OpenTagFile`.

`FieldMapName` is the name of the Fieldmap desired.

Return

Return	Description
`ADK_INVALID_HANDLE_VALUE`	On any error, such as invalid handle in argument, malloc problems, or not finding Fieldmap.
Anything else	Success

Example

```
#include “adkfns.h”
#include “adktypes.h”
...
ADK_CFG_HANDLE fHandle, sHandle;
char FileName[256];
...
fHandle = eLA_OpenTagFile(FileName);
if(fHandle != ADK_INVALID_HANDLE_VALUE)
  sHandle = eLA_GetFieldMap(fHandle, "Map1");
...
eLA_CloseTagHandle(sHandle);
eLA_CloseTagFile(fHandle);
```

eLA_GetFirstField

`eLA_GetFirstField` retrieves the information for the first line in a fieldmap section. This data is parsed into the appropriate structure members, up to the maximum field widths, with no validity checking.

Prototype

```
long eLA_GetFirstField(ADK_CFG_HANDLE MapHandle, FIELDMAP * FieldMap)
```

where

`MapHandle` is the handle returned by `eLA_GetFieldMap`.

`FieldMap` is a pointer to a FIELDMAP (typedef(d) struct).

Return

Return	Description
`ADK_INVALID_HANDLE_VALUE`	Invalid handle in argument.
`ADK_ERROR_NO_FIELDS`	No field lines for map.

Example

```
#include “adkfns.h”
#include “adktypes.h”
...
ADK_CFG_HANDLE fHandle, sHandle;
FIELDMAP FieldMap;
char FileName[256];
long rc;
...
fHandle = eLA_OpenTagFile(FileName);
if(fHandle != ADK_INVALID_HANDLE_VALUE)
  { sHandle = eLA_GetFieldMap(fHandle, “Map1”);
    if(sHandle != ADK_INVALID_HANDLE_VALUE)
    rc = eLA_GetFirstField(sHandle, &FieldMap);
    ...
  }
...
eLA_CloseTagHandle(sHandle);
eLA_CloseTagFile(fHandle);
```

eLA_GetNextField

`eLA_GetNextField` retrieves the information for successive lines in a fieldmap section. This data is parsed into the appropriate structure members, up to the maximum field widths, with no validity checking.

Prototype

```
long eLA_GetNextField(ADK_CFG_HANDLE MapHandle, FIELDMAP * FieldMap)
```

where

`MapHandle` is the handle returned by `eLA_GetFieldMap`.

`FieldMap` is a pointer to a FIELDMAP (typedef(d) struct).

Return

Return	**Description**
`ADK_INVALID_HANDLE_VALUE`	Invalid handle in argument.
`ADK_ERROR_LAST_FIELD`	No more fields for map.

Example

```
#include “adkfns.h”
#include “adktypes.h”
...
ADK_CFG_HANDLE fHandle, sHandle;
FIELDMAP FieldMap;
char FileName[256];
long rc;
...
fHandle = eLA_OpenTagFile(FileName);
if(fHandle != ADK_INVALID_HANDLE_VALUE)
  { sHandle = eLA_GetFieldMap(fHandle, “Map1”);
    if(sHandle != ADK_INVALID_HANDLE_VALUE)
      { rc = eLA_GetFirstField(sHandle, &FieldMap);
        while(rc == ADK_SUCCESS)
        { ...
          rc = eLA_GetNextField(sHandle, &FieldMap);
          ...
        }
      }
    ...
  }
...
```

```
eLA_CloseTagHandle(sHandle);
eLA_CloseTagFile(fHandle);
```

Hash Table API

Following are the hash table API functions used by the ADK.

eLA_InitHashTable

`eLA_InitHashTable` creates a hash table and returns a pointer to it. This table should be free(d) when no longer needed by eLA_DestHashTable.

Prototype

```
struct nlist * * eLA_InitHashtable(void);
```

Return

Return	Description
`NULL`	malloc error.
Anything else	Table address.

Example

```
#include “adkfns.h”
...
struct nlist * * cb_hash;
...
cb_hash = eLA_InitHashtable();
if(cb_hash == NULL)
  printf(“unable to make hash\n”);
else
...
...
...
eLA_DestHashtable(cb_hash);
```

eLA_DestHashTable

`eLA_DestHashTable` frees all of the dynamic memory in a hash table. It will behave intelligently when fed a NULL pointer.

Prototype

```
void eLA_DestHashtable(struct nlist * hashtable[])
```

Return

No returns.

Example

```
#include "adkfns.h"
...
struct nlist * * cb_hash;
...
cb_hash = eLA_InitHashtable();
...
...
...
eLA_DestHashtable(cb_hash);
```

eLA_put

`eLA_put` adds a new element to the hash table. It signals an error when a duplicate key is encountered.

Prototype

```
struct nlist * eLA_put(struct nlist * hashtable[], char * key,
      void * data, size_t datalen)
```

Return

Return	Description
`NULL`	Duplicate key or malloc error.
Anything else	Pointer to element (success).

Example

```
#include "adkfns.h"
...
struct nlist * * cb_hash, * element;
char data[81], key[81];
...
cb_hash = eLA_InitHashtable();
if(cb_hash == NULL)
```

```
  printf("unable to make hash\n");
else
  { strcpy(key, “passphrase”);
    strcpy(data, “Open Sesame”);
    element = eLA_put(cb_hash, key, data, 1 + strlen(data));
    if(element == NULL)
      printf(“unable to add element to hash table\n”);
    else
      ...
  }
...
...
eLA_DestHashtable(cb_hash);
```

eLA_get

`eLA_get` retrieves an element from the hash table. Note that a pointer to the element is returned - you must extract the data explicitly.

Prototype

```
struct nlist * eLA_get(struct nlist * hashtable[], char * key)
```

Return

Return	Description
`NULL`	Key not found.
Anything else	Pointer to element (success).

Example

```
#include “adkfns.h”
...
struct nlist * * cb_hash, * element;
char data[81], key[81];
...
cb_hash = eLA_InitHashtable();
if(cb_hash == NULL)
  printf(“unable to make hash\n”);
else
  { strcpy(key, “passphrase”);
    strcpy(data, “Open Sesame”);
    element = eLA_put(cb_hash, key, data, 1 + strlen(data));
    if(element == NULL)
      printf(“unable to add element to hash table\n”);
    else
```

```
        ...
    }

strcpy(key, “passphrase”);
element = eLA_get(cb_hash, key);
if(element == NULL)
  printf(“unable to fetch element from hash table\n”);
else
  ...
  ...
...
eLA_DestHashtable(cb_hash);
```

eLA_hash

Returns the hash value for a given key.

Prototype

```
unsigned int eLA_hash (char * key)
```

Return

Return	**Description**
Any value	Hash value for key.

Example

```
#include “adkfns.h”
...
unsigned int cbhash;
char key[81];
...
strcpy(key, “corned beef”);
cbhash = eLA_hash(key);
printf(“hash value for key = %s is %d\n”, key, cbhash);
...
...
```

Utility Functions and Macros

Following are the utility functions and macros used by the ADK.

eLA_catentry

`eLA_catentry` retrieves a specified error message from a message catalog file using the handle returned by `eLA_OpenCatalogFile`.

Prototype

```
char * eLA_catentry(char * msgbuffer, size_t bufferlen,
      ADK_CAT_HANDLE chandle, int msgnumber)
```

where

`msgbuffer` is the caller supplied buffer. This is filled to a maximum of bufferlen -1 chars with the message requested.

`bufferlen` is the size of the buffer.

`chandle` is the ADK_CAT_HANDLE returned by eLA_OpenCatalogFile.

`msgnumber` is the ID of the message desired.

Return

A message buffer is ALWAYS returned. This buffer may contain the following text

Return	**Description**
Invalid ADK_CAT_HANDLE	An invalid handle was passed.
Message xxx not found	Message ID xxx not found in file.

Example

```
#include adkfns.h
#include adktypes.h
...
ADK_CAT_HANDLE rHandle;
char catFileName[MAX_FNAME];
char msgbuffer[1024];
...
strcpy(catFileName, “c:\\tuxedo\elink\catalogs\ouradapter.text”
rHandle = eLA_OpenCatalogFile(catFileName);
...
eLA_log(eLA_catentry(msgbuffer, sizeof(msgbuffer), rHandle, 121);
...
eLA_CloseCatalogFile(rHandle);
```

eLA_chkeLinkLic

eLA_chkeLinkLic checks for a valid, current 'eLink Platform' section within the Tuxedo License File (TUXDIR/udataobj/lic.txt), in addition to checking the adapter_section specified in the argument to the function. Version numbers in the License File >= function argument are acceptable. eLA_chkeLinkLic can be used to verify the platform license only by passing a NULL in the adapter_section parameter.

Prototype

```
int eLA_chkeLinkLic(const char * adapter_section, const char
      * current_version)
```

Return

Return	Description
`-1`	Invalid (missing, expired, etc.) license. (Details in userlog()).
`0`	Success

Example

```
#include “adkfns.h”
...
int rc;
char Section[81], Version[81];
strcpy(Section, “eLink Adapter for PeopleSoft”);
strcpy(Version, “1.1”);
rc = eLA_chkeLinkLic(Section, Version);
printf(“eLinkLicense test, (v%s) - rc = %d\n”, Version, rc);
...
```

eLA_CloseCatalogFile

`eLA_CloseCatalogFile` closes a handle returned by `eLA_OpenCatalogFile` and frees any internally allocated resources. It can be called with an invalid handle.

Prototype `int eLA_CloseCatalogFile(ADK_CAT_HANDLE CatFileHandle)`

where

`CatFileHandle` is the handle returned by `eLA_OpenCatalogFile`.

Return

Return	**Description**
`ADK_SUCCESS`	Success
Anything else	Error

Example

```
#include adkfns.h
#include adktypes.h
...
ADK_CAT_HANDLE rHandle;
char catFileName[MAX_FNAME];
...
strcpy(catFileName, “c:\\tuxedo\elink\catalogs\tuxnt.text”
rHandle = eLA_OpenCatalogFile(catFileName);
...
eLA_CloseCatalogFile(rHandle);
```

eLA_GetConfigFileName

`eLA_GetConfigFileName` extracts the server configuration file name from the command line arguments passed to `tpsvrinit()` by Tuxedo.

Prototype

```
int eLA_GetConfigFileName(char * FileNameBuffer, size_t
       BufferSize,int argc, char * argv[])
```

where

`FileNameBuffer` is the caller-supplied buffer to receive the file name.

`BufferSize` is the length of the buffer.

`argc, argv` `argc,` is the argv as passed to `tpsvrinit()` by Tuxedo.

Return

Return	**Description**
`-1`	Buffer not large enough.
`0`	File Name parameter NOT found.
`>0`	Number of characters returned, including NULL.

Example

```
char configFileName[MAX_FNAME];
...
rc = eLA_GetConfigFileName(configFileName, MAX_FNAME, argc, argv);
printf(“eLA_GetConfigFileName - rc = %d\n”, rc);
if(rc == -1)
  printf(“Buffer not large enough\n”);
else if(rc == 0)
  printf(“File Name parameter not found\n”);
else
  printf(“File Name = %s\n”, configFileName);
```

eLA_hexdump

`eLA_hexdump` prints (to ULOG) data in the format given below, starting with the input address and continuing for maxchars bytes. If offlag = = ELA_HEX_ADDRESS_OFFSET, the offset from starting address is printed, otherwise the absolute address is printed.

Prototype

```
void eLA_hexdump(void * char_buffer, size_t maxchars, int offlag)
```

Example of Output

```
0012FB38  ff ff ff ff 54 fb 12 00   f1 85 f8 77 18 07 14 00   ....Tû..ñ..w....
0012FB48  74 0f 14 00 dc 05 14 00   09 00 00 00 00 00 00 00   t...Ü...........
0012FB58  09 00 00 00 b0 2b 00 10   50 0f 14 00 cc 05 14 01   ....°+..P.......
0012FB68  7d 1e f6 77 68                                      }..wh
```

eLA_log

`eLA_log` currently is a cover function for userlog(). It is intended that this function will be extended to handle message catalogs in the near future.

Prototype

```
void eLA_log(char *userText, ...)
```

Return None

Example

```
#include "adkfns.h"
...
eLA_log("Required parameter -C missing");
...
```

eLA_OpenCatalogFile

`eLA_OpenCatalogFile` opens a message catalog file (in text form), reads the information into memory, does some preliminary processing (for example, removes double quotes) and indexes the file. The file is closed and a handle to the data is returned. This handle is used by the `eLA_catentry` function and should be closed using `eLA_CloseCatalogFile` when no longer needed.

Prototype

```
ADK_CAT_HANDLE eLA_OpenCatalogFile(char * CatFileName)
```

where

`CatFileName` is the FULLY QUALIFIED file name for the message catalog.

Return

Return	Description
ADK_INVALID_HANDLE_VALUE	On any error, such as invalid handle in argument, malloc problems, or not finding Fieldmap.
Anything else	Success

Example

```
#include adkfns.h
#include adktypes.h
...
ADK_CAT_HANDLE rHandle;
```

```
char catFileName[MAX_FNAME];
...
strcpy(catFileName, “c:\\tuxedo\elink\catalogs\tuxnt.text”
rHandle = eLA_OpenCatalogFile(catFileName);
...
eLA_CloseCatalogFile(rHandle);
```

eLA_SetServerMsgLevel

`eLA_SetServerMsgLevel` extracts the min and max message levels from the values for the `MINMSGLEVEL` and `MAXMSGLEVEL` tags in the SERVER section of the configuration file.

Prototype

```
int eLA_SetServerMsgLevel(char * Filename, MSG_LEVEL * msglevels)
```

where

`Filename` is the configuration file name.

`msglevels` is the MSG_LEVEL struct to receive level data.

Return

Return	**Description**
`ADK_SUCCESS`	Found and extracted both values
`ADK_ERROR_SECTION_NOT_FOUND`	SERVER section not found.
`ADK_ERROR_PROPERTY_NOT_FOUND`	One or both levels not found.
`ADK_ERROR_OPEN_READ`	Unable to read file.

Example

```
MSG_LEVEL zLevel = {0,0};
char configFileName[MAX_FNAME];
...
rc = eLA_GetConfigFileName(configFileName, MAX_FNAME, argc, argv);
if(rc == ADK_SUCCESS)
  { rc = eLA_SetServerMsgLevel(configFileName, &zLevel);
    printf(“SetServerMsgLevel - rc = %d\n”,rc);
    if(rc == ADK_SUCCESS)
```

```
        printf(“min, max = %d, %d\n”, zLevel.minMsgLevel,
              zLevel.maxMsgLevel);
  }

...
```

ELACATENTRY

ELACATENTRY is a cover macro for the eLA_catentry function.

Definition #define ELACATENTRY(u,v,x,y) eLA_catentry((u),(v),(x),(y))

Example

```
...
char cat_buffer[ELA_MAX_ERROR_MESSAGE];
  size_t cat_len;
...
cat_len = sizeof(cat_buffer);
  strcpy(catFileName, “c:\\tuxedo\elink\catalogs\ouradapter.text”
  rHandle = eLA_OpenCatalogFile(catFileName);

...
eLA_log( ELACATENTRY(cat_buffer, cat_len, rHandle, 2221));
```

ELAIFTRACE

ELAIFTRACE calls a bracketed {} set of code if LVL falls within a range defined by minMsgLevel and maxMsgLevel of the given MSG_LEVEL structure VAR.

Definition

```
#define ELAIFTRACE(VAR,LVL) if( ((LVL) >= (VAR).minMsgLevel)
      && \((LVL)<= (VAR).maxMsgLevel
```

Example ELAIFTRACE can be used to invoke the eLA_hexdump program:

```
...
MSG_LEVEL zLevel = {0,0};
  char configFileName[256];
  ...
  eLA_SetServerMsgLevel(configFileName, &zLevel);
...
ELAIFTRACE(zLevel, 3){ eLA_hexdump(buffer, sizeof(buffer), 1);}
```

ELATRACE

ELATRACE calls eLA_log with ARGS if LVL falls within a range defined by minMsgLevel and maxMsgLevel of the given MSG_LEVEL structure VAR.

Definition
```
#define ELATRACE(VAR, LVL, ARGS) if(((LVL) >= (VAR).minMsgLevel)
        && \ ((LVL) <= (VAR).maxMsgLevel)
```

Example
```
...
MSG_LEVEL zLevel = {0,0};
  char cat_buffer[ELA_MAX_ERROR_MESSAGE];
  size_t cat_len;
...
cat_len = sizeof(cat_buffer);
...
eLA_SetServerMsgLevel(configFileName, &zLevel);
strcpy(catFileName, “c:\\tuxedo\elink\catalogs\ouradapter.text”
rHandle = eLA_OpenCatalogFile(catFileName);
...
ELATRACE(zLevel, 3, (ELACATENTRY(cat_buffer, cat_len, rHandle,
        1221)));
```

Definitions and Typedefs

The following typedef and definitions are used by the configuration file processing functions:

```
typedef int ADK_CFG_Handle
```

Function	**Typedef**
`#define ADK_INVALID_HANDLE_VALUE`	-1
`#define ADK_SUCCESS`	0
`#define ADK_ERROR_LAST_SECTION`	1
`#define ADK_ERROR_NO_PROPERTIES`	2
`#define ADK_ERROR_LAST_PROPERTY`	3

Function	Typedef
`#define ADK_ERROR_NO_TAG`	4
`#define ADK_ERROR_NO_VALUE`	5
`#define ADK_ERROR_BUFFER_OVERFLOW`	6
`#define ADK_ERROR_PROPERTY_NOT_FOUND`	7
`#define ADK_ERROR_SECTION_NOT_FOUND`	8
`#define ADK_INVALID_HANDLE_VALUE`	-1
`#define ADK_SUCCESS`	0

The following definitions are used by the `eLA_GetConfigFileName` and `eLA_SetServerMsgLevel` utility functions:

Function	Typedef
`#define ADK_ERROR_FILE_NOT_FOUND`	21
`#define ADK_ERROR_OPEN_READ`	22
`#define ADK_ERROR_OPEN_WRITE`	23
`#define ADK_ERROR_ON_READ`	24
`#define ADK_ERROR_ON_WRITE`	25

The following definition is used by the hash table functions:

```
struct nlist

{
  struct nlist *next; /* next element in the linked list */
  char *key;    /* String that is hashed     */
  void *data;   /* Data stored in hash table */
};
```

The following definitions are used by the and eLA_SetServerMsgLevel, eLA_hexdump and other utility functions, and the tracing macros:

```
typedef struct
```

```
 { int minMsgLevel;
   int maxMsgLevel;
 } MSG_LEVEL;
#define ELA_MAX_ERROR_MESSAGE 1024
#define ELA_HEX_ADDRESS_ABSOLUTE 0x00000000
#define ELA_HEX_ADDRESS_OFFSET   0x00000001
```

The following typedef is used by the message catalogue functions:

```
typedef long ADK_CAT_HANDLE;
```

The following #define(s) and typedef(s) are used in the ADK functions to support FML32 Field Maps in the configuration files:

#define ADK_ERROR_NO_FIELDS	8
#define ADK_ERROR_LAST_FIELD	9
#define FM_AN_MAX	256
#define FM_FN_MAX	32
#define FM_IO_MAX	16
#define FM_FD_MAX	16

```
typedef struct
 { char ApplicationName[FM_AN_MAX];
   char FML32FieldName[FM_FN_MAX];
   char InputOutput[FM_IO_MAX];
   char FieldDesignator[FM_FD_MAX];
 } FIELDMAP;
```

B ATMI References

The information included in this section is excerpted from the Tuxedo Online Documentation. These are some of the most commonly used Tuxedo ATMI functions used for adapter development. For additional details and a complete list of Tuxedo functions and commands, see http://edocs.beasys.com/tuxedo/tux65/index.htm.

Following is an alphabetical list of the the functions described in this section.

Refer to
tpacall
tpadvertise
tpalloc
tpcall
tpcancel
tpfree
tpgetrply
tpinit
tprealloc
tpreturn
tpsvrdone
tpsvrinit
tpterm

Refer to
tptypes
tpunadvertise

Client Membership

tpinit

Function	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`) that 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`), or when the application wishes to supply its own buffer type switch (see `typesw`). 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`. The buffer should be freed using `tpfree(3)` after calling `tpinit()`. The `TPINIT` typed buffer structure includes the following members:

```
char      usrname[MAXIDENT+2];
char      cltname[MAXIDENT+2];
char      passwd[MAXIDENT+2];
char      grpname[MAXIDENT+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()` prints a warning in a log file, ignores the setting and returns 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` 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` 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.

tpterm

Function 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, thenthe 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.

Buffer Management

tpalloc

Function 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` 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`, `realloc`, or `free` in the C library (for example, a buffer allocated with `tpalloc()` should not be freed with `free()`). Two buffer types are supported by any compliant implementation of the BEA Tuxedo system extension.

tprealloc

Function

Routine to change the size of a typed buffer

Synopsis

```
#include <atmi.h>
char * tprealloc(char *ptr, long size)
```

Description

`tprealloc()` changes the size of the buffer pointed to by `ptr` to size bytes and returns a pointer to the new (possibly moved) buffer. Similar to `tpalloc`, the size of the buffer will be at least as large as the larger of `size` and `dfltsize`, where `dfltsize` is the default buffer size specified in `tmtype_sw`. If the larger of the two is less than or equal to zero, then the buffer is unchanged and `NULL` is returned. A buffer's `type` remains the same after it is 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`).

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

tpfree

Function: Routine for freeing a typed buffer

Synopsis:

```
#include <atmi.h>
void tpfree(char *ptr)
```

Description: The argument to `tpfree()` is a pointer to a buffer previously obtained by either `tpalloc` or `tprealloc`. If `ptr` is `NULL`, no action occurs. Undefined results will occur if `ptr` does not point to a typed buffer (or if it points to space previously freed with `tpfree()`). Inside service routines, `tpfree()` returns and does not free the buffer if `ptr` points to the buffer passed into a service routine. Some buffer types require state information or associated data to be removed as part of freeing a buffer. `tpfree()` removes any of these associations (in a communication manager-specific manner) before a buffer is freed. Once `tpfree()` returns, `ptr` should not be passed as an argument to any BEA Tuxedo 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`, `realloc` or `free` in the C library (for example, a buffer allocated with tpalloc should not be freed with free).

tptypes

Function: Routine to determine information about a typed buffer

Synopsis

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

Description

`tptypes()` takes as its first argument a pointer to a data buffer and returns the `type` and `subtype` of that buffer in its second and third arguments, respectively. `ptr` must point to a buffer gotten from `tpalloc`. If `type` and `subtype` are non-NULL, then the function populates the character arrays to which they point with the names of the buffer's `type` and `subtype`, respectively. If the names are of their maximum length (8 for `type`, 16 for `subtype`), the character array is not null-terminated. If no `subtype` exists, then the array pointed to by `subtype` will contain a NULL string. Note that only the first eight bytes of `type` and the first 16 bytes of `subtype` are populated.

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`).

`[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.

Request/Response

tpcall

Function

Routine for sending service request and awaiting its reply

Synopsis

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

Description

tpcall sends a request and synchronously awaits its reply. A call to this function is the same as calling tpacall immediately followed by tpgetrply. tpcall sends a request to the service named by svc. The request is sent out at the priority defined for svc unless overridden by a previous call to tpsprio. The data portion of a request is pointed to by idata, a buffer previously allocated by tpalloc. ilen specifies how much of idata to send. Note that if idata points to a buffer of a type that does not require a length to be specified, (for example, an FML fielded buffer), then ilen is ignored (and may be 0). Also, idata may be NULL, in which case ilen is ignored. The type and 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 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 tpgetrply was issued with *len. 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`. `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` with no intermediate ATMI calls, may provide more detailed information about the generated error. Refer to the `tperrordetail` 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 inflags 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 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` or `tpforward` (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.

tpacall

Function
: 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`. If data is non-NULL, it must point to a buffer previously allocated by `tpalloc` and `len` should specify the amount of data in the buffer that should be sent. Note that if data points to a buffer of a type that does not require a length to be specified, (for example, an FML fielded buffer), then `len` is ignored (and may be 0). If `data` is NULL, `len` is ignored and a request is sent with no data portion. The `type` and `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.

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`, 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.

tpgetrply

Function Routine for getting a reply from a previous request

Synopsis

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

Description `tpgetrply` returns a reply from a previously sent request. This function's first argument, `cd`, points to a call descriptor returned by `tpacall`. By default, the function waits until the reply matching `*cd` arrives or a timeout occurs. `data` must be the address of a pointer to a buffer previously allocated by `tpalloc` and `len` should point to a long that `tpgetrply` sets to the amount of data successfully received. Upon successful return, `*data` points to a buffer containing the reply and `*len` contains the size of the data. FML and FML32 buffers often assume a minimum size of 4096 bytes; if the reply is larger than 4096, the size of the buffer is increased to a size large enough to accommodate the data being returned. 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`. 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` 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` with no intermediate ATMI calls, may provide more detailed information about the generated error. Refer to the `tperrordetail` 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` 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.

tpcancel

Function Routine for canceling a call descriptor for outstanding reply

Synopsis

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

Description `tpcancel()` cancels a call descriptor, `cd`, returned by `tpacall`. It is an error to attempt to cancel a call descriptor associated with a transaction. Upon success, `cd` is no longer valid and any reply received on behalf of `cd` will be silently discarded.

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.

Service Entry and Return

tpsvrinit

Function 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). The options are passed through `argc` and `argv`. Since `getopt` 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.

tpsvrdone

Function
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` or `tpforward` is called in `tpsvrdone`, it simply returns having no effect.

tpreturn

Funtion

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`, `tpgetrply`, or `tprecv`, then after a successful call to `tpreturn`, the reply is available in the receiver's buffer. For conversational services, `tpreturn` also tears down the connection. That is, the service routine cannot call `tpdiscon` directly. To ensure correct results, the program that connected to the conversational service should not call `tpdiscon`; rather, it should wait for notification that the conversational service has completed (that is, it should wait for one of the events, like `TPEV_SVCSUCC` or `TPEV_SVCFAIL`, sent by `tpreturn`). If the service routine was in transaction mode, `tpreturn` places the service's portion of the transaction in a state 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` or `tpabort` is called by the originator of the transaction. `tpreturn` should be called after receiving all replies expected from service requests initiated by the service routine. Otherwise, depending on the nature of the service, either a `TPESVCERR` status or `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`.

Arguments

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 applicatio-defined return code, `rcode`, may be sent to the program receiving the service reply. This code is sent regardless of the setting of `rval` as long as a reply can be successfully sent (that is, as long as the receiving call returns success or `TPESVCFAIL`). In addition, for conversational services, this code can be sent only if the service routine has control of the connection when it issues `tpreturn`. The value of `rcode` is available in the receiver in the variable, `tpurcode`. `data` points to the data portion of a reply to be sent. If data is non-NULL, it must point to a buffer previously obtained by a call to `tpalloc`. If this is the same buffer passed to the service routine upon its invocation, then its disposition is up to the BEA Tuxedo 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`. A conversational service routine must use `tpreturn`, and cannot use `tpforward`. If a service routine returns without using either `tpreturn` or `tpforward` (that is, it uses the C language return statement or just simply "falls out of the function") or `tpforward` is called from a conversational server, the server will print a warning message in the log and return a service error to the service requester. In addition, all open connections to subordinates will be disconnected immediately, and any outstanding asynchronous replies will be dropped. If the server was in transaction mode at the time of failure, the transaction is marked rollback-only. Note also that if either `tpreturn` or `tpforward` are used outside of a service routine (for example, in clients, or in `tpsvrinit` or `tpsvrdone`), then these routines simply return having no effect.

Errors Since `tpreturn` ends the service routine, any errors encountered either in handling arguments or in processing cannot be indicated to the function's caller. Such errors cause `tperrno` to be set to `TPESVCERR` for a program receiving the service's outcome via either `tpcall` or `tpgetrply`, and cause the event, `TPEV_SVCERR`, to be sent over the conversation to a program using `tpsend` or `tprecv`. If either `SVCTIMEOUT` in the `ubbconfig` file or `TA_SVCTIMEOUT` in the `TM_MIB` is non-zero, the event `TPEV_SVCERR` is returned when a service timeout occurs. `tprrordetail` and `tpstrerrordetail` 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.

Dynamic Advertisement

tpadvertise

Function
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 (""). `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 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.

`[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.

tpunadvertise

Function

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`). 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.

C FML32 API

The following information is excerpted from the Tuxedo Online Documentation. These are some of the most commonly used Tuxedo FML32 API functions used for adapter development. For additional details and a complete list of Tuxedo functions and commands, see http://edocs.beasys.com/tuxedo/tux65/index.htm.

Following is an alphabetical list of the the functions described in this section.

Refer to
Fadd, Fadd32
Fchg, Fchg 32
Ffind, Ffind32
Fget, Fget32
Fielded, Fielded32
Finit, Finit32
Fldid, Fldid32
Fneeded, Fneeded32
Fsizeof, Fsizeof32

Fadd, Fadd32

Function
: Add new field occurrence

Synopsis

```
#include stdio.h>
#include "fml.h"
int Fadd(FBFR *fbfr, FLDID fieldid, char *value, FLDLEN len)
#include "fml32.h"
int Fadd32(FBFR32 *fbfr, FLDID32 fieldid, char *value, FLDLEN32
len)
```

Description

`Fadd()` adds the specified field value to the given buffer. `fbfr` is a pointer to a fielded buffer. `fieldid` is a field identifier. `value` is a pointer to a new value; the pointer's type must be the same `fieldid type` as the value to be added. `len` is the length of the value to be added; it is required only if type is `FLD_CARRAY` The value to be added is contained in the location pointed to by the `value` parameter. If one or more occurrences of the field already exist, then the value is added as a new occurrence of the field, and is assigned an occurrence number 1 greater than the current highest occurrence (to add a specific occurrence, `Fchg` must be used). In the SYNOPSIS section above the value argument to `Fadd()` is described as a character pointer data type (`char *` in C). Technically, this describes only one particular kind of value passable to `Fadd()`. In fact, the type of the value argument should be a pointer to an object of the same type as the type of the fielded-buffer representation of the field being added. For example, if the field is stored in the buffer as type `FLD_LONG`, then `value` should be of type pointer-to-long (`long *` in C). Similarly, if the field is stored as `FLD_SHORT`, then `value` should be of type pointer-to-short (short * in C). The important thing is that `Fadd()` assumes that the object pointed to by value has the same `type` as the stored `type` of the field being added. For values of type `FLD_CARRAY`, the length of the value is given in the `len` argument.For all `types` other than `FLD_CARRAY`, the length of the object pointed to by `value` is inferred from its type (e.g. a value of type `FLD_FLOAT` is of length `sizeof(float)`), and the contents of `len` are ignored. Fadd32 is used with 32-bit FML.

Return Values

This function returns `-1` on error and sets `Ferror` to indicate the error condition.

Errors

Under the following conditions, `Fadd()` fails and sets `Ferror` to:

`[FALIGNERR] "fielded buffer not aligned"`
: The buffer does not begin on the proper boundary.

`[FNOTFLD] "buffer not fielded"`
: The buffer is not a fielded buffer or has not been initialized by `Finit()`.

`[FEINVAL] "invalid argument to function"`
: One of the arguments to the function invoked was invalid. (For example, specifying a NULL value parameter to `Fadd`.)

`[FNOSPACE] "no space in fielded buffer"`
: A field value is to be added in a fielded buffer but there is not enough space remaining in the buffer.

`[FBADFLD] "unknown field number or type"`
: A field number is specified which is not valid.

Fchg, Fchg 32

Function
: Change field occurrence value

Synopsis

```
#include <stdio.h>
#include "fml.h"
int
Fchg(FBFR *fbfr, FLDID fieldid, FLDOCC oc, char *value, FLDLEN len)
#include "fml32.h"
int
Fchg32(FBFR32 *fbfr, FLDID32 fieldid, FLDOCC32 oc, char *value,
FLDLEN32 len)
```

Description
: `Fchg()` changes the value of a field in the buffer. `fbfr` is a pointer to a fielded buffer. `fieldid` is a field identifier. `oc` is the occurrence number of the field. `value` is a pointer to a new value, its type must be the same type as the value to be changed (see below). `len` is the length of the value to be changed; it is required only if field type is `FLD_CARRAY`. If an occurrence of `-1` is specified, then the field value is added as a new occurrence to the buffer. If the specified field occurrence is found, then the field value is modified to the value specified. If a field occurrence is specified that does not exist, then NULL values are added for the missing occurrences until the desired occurrence can be added (for example, changing field occurrence 4 for a field that does not exist on a buffer will cause 3 NULL values to be added followed by the specified field value). NULL values consist of the NULL string (1 byte in length) for string and character values, 0 for long and short fields, 0.0 for float and double values, and a zero-length string for a character array. The new or modified value is contained in `value` and its length is given in `len` if it is a character array (ignored in other cases). If value is NULL, then the field occurrence is deleted. A value to be deleted that is not found, is considered an error. In the SYNOPSIS section above the value argument to `Fchg()`

is described as a character pointer data type (char * in C). Technically, this describes only one particular kind of value passable to `Fchg()`. In fact, the type of the value argument should be a pointer to an object of the same type as the type of the fielded-buffer representation of the field being changed. For example, if the field is stored in the buffer as type `FLD_LONG`, then `value` should be of type `pointer-to-long` (long * in C). Similarly, if the field is stored as `FLD_SHORT`, then `value` should be of type `pointer-to-short` (short * in C). The important thing is that `Fchg()` assumes that the object pointed to by value has the same type as the stored type of the field being changed.

Fchg32 is used with 32-bit FML.

Return Values This function returns `-1` on error and sets `Ferror` to indicate the error condition.

Errors Under the following conditions, `Fchg()`fails and sets `Ferror` to:

`[FALIGNERR] "fielded buffer not aligned"`
The buffer does not begin on the proper boundary.

`[FNOTFLD] "buffer not fielded"`
The buffer is not a fielded buffer or has not been initialized by `Finit()`.

`[FNOTPRES] "field not present"`
A field occurrence is requested for deletion but the specified field and/or occurrence was not found in the fielded buffer.

`[FNOSPACE] "no space in fielded buffer"`
A field value is to be added or changed in a fielded buffer but there is not enough space remaining in the buffer.

`[FBADFLD] "unknown field number or type"`
A field identifier is specified which is not valid.

Ffind, Ffind32

Function Find field occurrence in buffer

Synopsis
```
#include <stdio.h>
#include "fml.h"
char *
Ffind(FBFR *fbfr, FLDID fieldid, FLDOCC oc, FLDLEN *len)
```

```
#include "fml32.h"
char *
Ffind32(FBFR32 *fbfr, FLDID32 fieldid, FLDOCC32 oc, FLDLEN32 *len)
```

Description

`Ffind()` finds the value of the specified field occurrence in the buffer. `fbfr` is a pointer to a fielded buffer. `fieldid` is a field identifier. `oc` is the occurrence number of the field. If the field is found, its length is set into `*len`, and its location is returned as the value of the function. If the value of `len` is NULL, then the field length is not returned. `Ffind()` is useful for gaining read-only access to a field. In no case should the value returned by `Ffind()` be used to modify the buffer. In general, the locations of values of types `FLD_LONG`, `FLD_FLOAT`, and `FLD_DOUBLE` are not suitable for direct use as their stored type, since proper alignment within the buffer is not guaranteed. Such values must be copied first to a suitably aligned memory location. Accessing such fields through the conversion function `CFfind` does guarantee the proper alignment of the found converted value. Buffer modification should only be done by the functions `Fadd` or `Fchg`. The values returned by `Ffind()` and `Ffindlast()` are valid only so long as the buffer remains unmodified.

Ffind32 is used with 32-bit FML.

Return Values

In the SYNOPSIS section above, the return value to `Ffind()` is described as a character pointer data type (char* in C). Actually, the pointer returned points to an object that has the same type as the stored type of the field. This function returns a pointer to NULL on error and sets `Ferror` to indicate the error condition.

Errors

Under the following conditions, `Ffind()` fails and sets `Ferror` to:

`[FALIGNERR] "fielded buffer not aligned"`
The buffer does not begin on the proper boundary.

`[FNOTFLD] "buffer not fielded"`
The buffer is not a fielded buffer or has not been initialized by `Finit()`.

`[FNOTPRES] "field not present"`
A field occurrence is requested but the specified field and/or occurrence was not found in the fielded buffer.

`[FBADFLD] "unknown field number or type"`
A field identifier is specified which is not valid.

Fget, Fget32

Function

Get copy and length of field occurrence

Synopsis

```
#include <stdio.h>
#include "fml.h"
int
Fget(FBFR *fbfr, FLDID fieldid, FLDOCC oc, char *value, FLDLEN
    *maxlen)
#include "fml32.h"
int
Fget32(FBFR32 *fbfr, FLDID32 fieldid, FLDOCC32 oc, char
*value,FLDLEN32 *maxlen)
```

Description

`Fget()` should be used to retrieve a field from a fielded buffer when the value is to be modified. `fbfr` is a pointer to a fielded buffer. `fieldid` is a field identifier. `oc` is the occurrence number of the field. The caller provides `Fget()` with a pointer to a private data area, `loc`, as well as the length of the data area, `*maxlen`, and the length of the field is returned in `*maxlen`. If `*maxlen` is NULL when the function is called, then it is assumed that the data area for the field value `loc` is big enough to contain the field value and the length of the value is not returned. If `loc` is NULL, the value is not retrieved. Thus, the function call can be used to determine the existence of the field.

In the SYNOPSIS section above the value argument to `Fget()` is described as a character pointer data type (char * in C). Technically, this describes only one particular kind of value passable to `Fget()`. In fact, the type of the value argument should be a pointer to an object of the same type as the type of the fielded-buffer representation of the field being retrieved. For example, if the field is stored in the buffer as type `FLD_LONG`, then value should be of type `pointer-to-long` (long * in C). Similarly, if the field is stored as `FLD_SHORT`, then value should be of type `pointer-to-short` (short * in C). The important thing is that `Fget()` assumes that the object pointed to by value has the same type as the stored type of the field being retrieved.

Fget32 is used with 32-bit FML.

Return Values

This function returns `-1` on error and sets `Ferror` to indicate the error condition.

Errors

Under the following conditions, `Fget()` fails and sets `Ferror` to:

`[FALIGNERR] "fielded buffer not aligned"`
    The buffer does not begin on the proper boundary.

`[FNOTFLD] "buffer not fielded"`
The buffer is not a fielded buffer or has not been initialized by `Finit()`.

`[FNOSPACE] "no space"`
The size of the data area, as specified in `maxlen`, is not large enough to hold the field value.

`[FNOTPRES] "field not present"`
A field occurrence is requested but the specified field and/or occurrence was not found in the fielded buffer.

`[FBADFLD] "unknown field number or type"`
A field identifier is specified which is not valid.

Fielded, Fielded32

Function
Return true if buffer is fielded

Synopsis

```
#include stdio.h>
#include "fml.h"
int
Fielded(FBFR *fbfr)
#include "fml32.h"
int
Fielded32(FBFR32 *fbfr)
```

Description
`Fielded()` is used to test whether the specified buffer is fielded. `fbfr` is a pointer to a fielded buffer.

Fielded32 is used with 32-bit FML.

Return Values
`Fielded()` returns true `(1)` if the buffer is fielded. It returns false (0) if the buffer is not fielded and does not set `Ferror` in this case.

Finit, Finit32

Function
Initialize fielded buffer

Synopsis

```
#include <stdio.h>
#include "fml.h"
int
Finit(FBFR *fbfr, FLDLEN buflen)
#include "fml32.h"
int
Finit32(FBFR32 *fbfr, FLDLEN32 buflen)
```

Description

`Finit()`can be called to initialize a fielded buffer statically. `fbfr` is a pointer to a fielded buffer. `buflen` is the length of the buffer. The function takes the buffer pointer and buffer length, and sets up the internal structure for a buffer with no fields. `Finit()` can also be used to re-initialize a previously used buffer.

Finit32 is used with 32-bit FML.

Return Values

This function returns `-1` on error and sets `Ferror` to indicate the error condition.

Errors

Under the following conditions, `Finit()` fails and sets `Ferror` to:

`[FALIGNERR] "fielded buffer not aligned"`
The buffer does not begin on the proper boundary.

`[FNOTFLD] "buffer not fielded"`
The buffer pointer is NULL.

`[FNOSPACE] "no space in fielded buffer"`
The buffer size specified is too small for a fielded buffer.

Example

The correct way to re-initialize a buffer to have no fields is:

```
Finit(fbfr,
(FLDLEN)Fsizeof(fbfr));
```

Fldid, Fldid32

Function

Map field name to field identifier

Synopsis

```
#include <stdio.h>
#include "fml.h"
FLDID
Fldid(char *name)
#include "fml32.h"
```

```
FLDID32
Fldid32(char *name)
```

Description

`Fldid()` provides a runtime translation of a field-name to its field identifier and returns a `FLDID` corresponding to its field name parameter. The first invocation causes space to be dynamically allocated for the field tables and the tables to be loaded. To recover data space used by the field tables loaded by `Fldid()`, the user may unload the files by a call to the `Fnmid_unload` function.

Fldid32 is used with 32-bit FML.

Return Values

This function returns `BADFLDID` on error and sets `Ferror` to indicate the error condition.

Errors

Under the following conditions, `Fldid()`fails and sets `Ferror` to:

`[FBADNAME] "unknown field name"`
A field name is specified which cannot be found in the field tables.

`[FMALLOC] "malloc failed"`
Allocation of space dynamically using `malloc(3)` failed.

Fneeded, Fneeded32

Function

Compute size needed for buffer

Synopsis

```
#include <stdio.h>
#include "fml.h"
long
Fneeded(FLDOCC F, FLDLEN V)
#include "fml32.h"

long
Fneeded32(FLDOCC32 F, FLDLEN32 V)
```

Description

`Fneeded()` if used to determine the space that must be allocated for `F` fields and `V` bytes of value space.

Fneeded32 is used with 32-bit FML.

Return Values

This function returns `\-1` on error and sets `Ferror` to indicate the error condition.

Errors

Under the following conditions, `Fneeded()` fails and sets `Ferror` to:

`[FEINVAL] "invalid argument to function"`
One of the arguments to the function invoked was invalid, (for example, number of fields is less than 0, `V` is 0 or total size is greater than 65534).

Fsizeof, Fsizeof32

Function

Return size of fielded buffer

Synopsis

```
#include "fml32.h"
long
Fsizeof32(FBFR32 *fbfr)
```

Description

`Fsizeof()` returns the size of a fielded buffer in bytes. `fbfr` is a pointer to a fielded buffer. `Fsizeof32` is used with 32-bit FML.

Return Values

This function returns `\-1` on error and sets `Ferror` to indicate the error condition.

Errors

Under the following conditions, `Fsizeof()` fails and sets `Ferror` to:

`[FALIGNERR] "fielded buffer not aligned"`
The buffer does not begin on the proper boundary.

`[FNOTFLD]"buffer not fielded"`
The buffer is not a fielded buffer or has not been initialized by `Finit()`.

Example of a Server that Uses FML32

The following server receives an FML32 buffer as the data field in a `TPSVRINFO` struct, deletes the contents of all of the fields, than repopulates them in the opposite order

Listing 0-1 Example of Server that Uses FML32

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

```
#include "atmi.h"
#include "fml32.h"
#include "userlog.h"

#define TEST_STRING "STRTEST"
#define TEST_LONG   70001
#define TEST_CHAR   'Z'
#define TEST_SHORT  911
#define TEST_CARRAY "TESTC"

FMLFOO(msg)
TPSVCINFO *msg;
{
        FBFR32 *fbfr;/* data to be sent */
        FLDLEN32 fbfr_len;
        FLDID32 fieldid;

        long  test_long;
        char  test_char;
        short test_short;

        fbfr = (FBFR32 *) msg->data;

        /*-----------------------------------*/
        /* Delete all fields in FML32 buffer */
        /*-----------------------------------*/
        fieldid = Fldid32("MYSTRING");
        if (Fdel32(fbfr, fieldid, 0) < 0)
        {
          userlog("Fdel32 MYSTRING failed:  ");
        }
       fieldid = Fldid32("MYLONG");
        if (Fdel32(fbfr, fieldid, 0) < 0)
        {
          userlog("Fdel32 MYLONG failed:  ");
        }
        fieldid = Fldid32("MYCHAR");
        if (Fdel32(fbfr, fieldid, 0) < 0)
        {
          userlog("Fdel32 MYCHAR failed:  ");
        }
        fieldid = Fldid32("MYSHORT");
        if (Fdel32(fbfr, fieldid, 0) < 0)
        {
          userlog("Fdel32 MYSHORT failed:  ");
        }
        fieldid = Fldid32("MYCARRAY");
        if (Fdel32(fbfr, fieldid, 0) < 0)
        {
          userlog("Fdel32 MYCARRAY failed:  ");
        }
        /*----------------------------------*/
        /* Add all fields to FML32 buffer in opposite order */
        /*----------------------------------*/
        fieldid = Fldid32("MYCARRAY");
      if (Fadd32(fbfr, fieldid, TEST_CARRAY, (FLDLEN32) sizeof(TEST_CARRAY)) < 0)

        {
          userlog("Fadd32 MYCARRAY failed:  ");
        }
        fieldid = Fldid32("MYSHORT");
        test_short = TEST_SHORT;
        if (Fadd32(fbfr, fieldid, (char *) &test_short, (FLDLEN32)
             sizeof(test_short)) < 0)
        {
          userlog("Fadd32 MYSHORT failed:  ");
        }
        fieldid = Fldid32("MYCHAR");
        test_char = TEST_CHAR;
        if (Fadd32(fbfr, fieldid, (char *) &test_char, (FLDLEN32)
             sizeof(test_char)) < 0)
        {
          userlog("Fadd32 MYCHAR failed:  ");
        }
        fieldid = Fldid32("MYLONG");
        test_long = TEST_LONG;
        if (Fadd32(fbfr, fieldid, (char *) &test_long, (FLDLEN32)
             sizeof(test_long)) < 0)
        {
          userlog("Fadd32 MYLONG failed:  ");
        }
        fieldid = Fldid32("MYSTRING");
         if (Fadd32(fbfr, fieldid, TEST_STRING, (FLDLEN32) strlen(TEST_STRING))
              < 0)

        {
          userlog("Fadd32 MYSTRING failed:  ");
        }
        tpreturn(TPSUCCESS, 0, msg->data, 0L, 0);
}
```

D Tuxedo Commands

The following information is excerpted from the Tuxedo Online Documentation. These are some of the most commonly used Tuxedo commands used for adapter development. For additional details and a complete list of Tuxedo functions and commands, see http://edocs.beasys.com/tuxedo/tux65/index.htm.

Following is an alphabetical list of the the commands described in this section.

Refer to
buildclient
buildserver
tmadmin
tmboot
tmloadcf
tmshutdown
ud, ud32, wud, wud32

buildclient

Function Construct a BEA Tuxedo client module

Synopsis
```
buildclient [ -C ] [ -v ] [ {-r rmname | -w } ] [ -o name] [ -f
firstfiles] [ -l lastfiles]
```

Description

`buildclient` is used to construct a BEA Tuxedo client module. The command combines the files supplied by the `-f` and `-l` options with the standard BEA Tuxedo libraries to form a load module. The load module is built by `buildclient` using the default C language compilation command defined for the operating system in use. The default C language compilation command for the UNIX System is the `cc` command described in UNIX System reference manuals.

`-v`

specifies that `buildclient` should work in verbose mode. In particular, it writes the compilation command to its standard output.

`-w`

specifies that the client is to be built using the workstation libraries. The default is to build a native client if both native mode and workstation mode libraries are available. This option cannot be used with the `-r` option.

`-r rmname`

specifies the resource manager associated with this client. The value `rmname` must appear in the resource manager table located in `$TUXDIR/udataobj/RM`. Each line in this file is of the form:

```
rmname:rmstructure_name:library_names
```

(See the buildtms command in the *BEA Tuxedo Reference Manual at* http://edocs.beasys.com/tuxedo/tux65/index.htm for further details.) Using the `rmname` value, the entry in `$TUXDIR/udataobj/RM` is used to include the associated libraries for the resource manager automatically and to set up the interface between the transaction manager and resource manager properly. The value `Tuxedo/D` includes the libraries for the Tuxedo System/D resource manager. The value `Tuxedo/SQL` includes the libraries for the Tuxedo System/SQL resource manager. Other values can be specified as they are added to the resource manager table. If the `-r` option is not specified, the default is that the client is not associated with a resource manager. Refer to the `ubbconfig` reference page.

`-o`

specifies the file name of the output load module. If not supplied, the load module is named `a.out`.

`-f`

specifies one or more user files to be included in the compilation and link edit phases of `buildclient first`, before the BEA Tuxedo libraries. If more than one file is specified, file names must be separated by white space and the entire list must be enclosed in quotation marks. This option may be specified multiple times. The `CFLAGS` and `ALTCFLAGS` environment variables, described below, should be used to include any compiler options and their arguments.

`-l`

specifies one or more user files to be included in the compilation and link edit phases of `buildclient last`, after the BEA Tuxedo libraries. If more than one file is specified, file names must be separated by white space and the entire list must be enclosed in quotation marks. This option may be specified multiple times.

`-C`

specifies COBOL compilation.

Environment Variables

`TUXDIR`

`buildclient` uses the environment variable `TUXDIR` to find the System/T libraries and include files to use during compilation of the client process.

`CC`

`buildclient` normally uses the default C language compilation command to produce the client executable. The default C language compilation command is defined for each supported operating system platform and is defined as `cc(1)` for UNIX System. In order to allow for the specification of an alternate compiler, `buildclient` checks for the existence of an environment variable named `CC`. If `CC` does not exist in buildclient's environment, or if it is the string "", `buildclient` will use the default C language compiler. If `CC` does exist in the environment, its value is taken to be the name of the compiler to be executed.

`CFLAGS`

The environment variable `CFLAGS` is taken to contain a set of arguments to be passed as part of the compiler command line. This is in addition to the command line option, `"-I${TUXDIR}/include"` passed automatically by `buildclient`. If `CFLAGS` does not exist in `buildclient’s` environment, or if it is the string, "", no compiler command line arguments are added by `buildclient`.

`ALTCC`

When the `-C` option is specified for COBOL compilation, `buildclient` normally uses the BEA Tuxedo shell `cobcc,` which in turn calls `cob` to produce the client executable. In order to allow for the specification of an alternate compiler, `buildclient` checks for the existence of an environment variable named `ALTCC`. If `ALTCC` does not exist in `buildclient’s` environment, or if it is the string, "", `buildclient` will use `cobcc`. If `ALTCC` does exist in the environment, its value is taken to be the name of the compiler command to be executed.

`ALTCFLAGS`

The environment variable `ALTCFLAGS` is taken to contain a set of additional arguments to be passed as part of the COBOL compiler command line when the `-C` option is specified. This is in addition to the command line option, `"-I${TUXDIR}/include”,` passed automatically by `buildclient`. When the `-C` option is used, putting compiler options and their arguments in the `buildclient -f` option will generate errors; they must be put in `ALTCFLAGS`. If not set, then the value is set to the same value used for `CFLAGS`, as specified above.

`COBOPT`

The environment variable `COBOPT` is taken to contain a set of additional arguments to be used by the COBOL compiler, when the `-C` option is specified.

`COBCPY`

The environment variable, `COBCPY,` indicates which directories contain a set of COBOL copy files to be used by the COBOL compiler when the `-C` option is specified.

`LD_LIBRARY_PATH`

The environment variable, `LD_LIBRARY_PATH` ,indicates which directories contain shared objects to be used by the COBOL compiler in addition to the BEA Tuxedo system shared objects.

Examples

```
CC=ncc CFLAGS="-I /APPDIR/include"; export CC CFLAGS
buildclient -o empclient -f emp.c -f "userlib1.a userlib2.a"
```

buildserver

Function

Construct a BEA Tuxedo server load module

Synopsis

```
buildserver [-C] [-s { @filename | service[,service...][:func] |
    :func } ] [-n maxdynam] [-v] [-o outfile] [-f firstfiles]
[-l lastfiles] [{-r|-g} rmname] [-k]
```

Description

`buildserver` is used to construct a BEA Tuxedo server load module. The command combines the files supplied by the `-f` and `-l` options with the standard server main routine and the standard BEA Tuxedo libraries to form a load module. The load module is built by the `cc(1)` command, which `buildserver` invokes. (See `cc` in any UNIX System reference manual.) The options to `buildserver` have the following meaning:

`-v`

specifies that `buildserver` should work in verbose mode. In particular, it writes the compilation command to its standard output.

`-o outfile`

specifies the name of the file the output load module is to have. If not supplied, the load module is named `SERVER`.

`-n maxdynam`

specifies the maximum number of dynamic services the user can specify when the server is run. A dynamic service allows the user to specify at run time the function within the server that is to process the service. If `-n` is not specified, the maximum number of such services is set to 25.

`-f firstfiles`

specifies one or more user files to be included in the compilation and link edit phases of `buildserver first`, before the BEA Tuxedo libraries. If more than one file is specified, file names must be separated by white space and the entire list must be enclosed in quotation marks. This option may be specified multiple times. The `CFLAGS` and `ALTCFLAGS` environment variables, described below, should be used to include any compiler options and their arguments.

`-l lastfiles`

specifies one or more user files to be included in the compilation and link edit phases of `buildserver last`, after the BEA Tuxedo libraries. If more than one file is specified, file names must be separated by white space and the entire list must be enclosed in quotation marks. This option may be specified multiple times.

`-r rmname`

specifies the resource manager associated with this server. The value rmname must appear in the resource manager table located in `$TUXDIR/udataobj/RM`. Each line in this file is of the form:

```
rmname:rmstructure_name:library_names
```

(See the `buildtms` command in the *BEA Tuxedo Reference Manual at* http://edocs.beasys.com/tuxedo/tux65/index.htm for further details.) Using the `rmname` value, the entry in `$TUXDIR/udataobj/RM` is used to include the associated libraries for the resource manager automatically and to set up the interface between the transaction manager and resource manager properly. The value, `Tuxedo/D` includes the libraries for the BEA Tuxedo System/D resource manager. The value, `Tuxedo/SQL` includes the libraries for the BEA Tuxedo System/SQL resource manager. Other values can be specified as they are added to the resource manager table. If the `-r` option is not specified, the default is to use the null resource manager. Refer to the `ubbconfig` reference page

```
-s { @filename | service[,service...][:func] | :func } ]
```

specifies the names of services that can be advertised when the server is booted. Service names (and implicit function names) must be less than or equal to 15 characters in length. An explicit function name (that is, a name specified after a colon) can be up to 128 characters in length. Names longer than these limits are truncated with a warning message. When retrieved by `tmadmin` or `TM_MIB`, only the first 15 characters of a name are displayed. (See `servopts(5)`.) All functions that can be associated with a service must be specified with this option. In the most common case, a service is performed by a function that carries the same name; that is, the x service is performed by function x. For example, the specification

```
-s x,y,z
```

will build the associated server with services x, y, and z, each to be processed by a function of the same name. In other cases, a service (or several services) may be performed by a function of a different name. The specification

```
-s x,y,z:abc
```

builds the associated server with services x, y, and z, each to be processed by the function abc. Spaces are not allowed between commas. Function name is preceded by a colon. In another case, the service name may not be known until runtime. Any function that can have a service associated with it must be specified to `buildserver`. To specify a function that can have a service name mapped to it, put a colon in front of the function name. For example, the specification

`-s :pqr`

builds the server with a function `pqr`, which can have a service association. `Tpadvertise` could be used to map a service name to the `pqr` function. A filename can be specified with the `-s` option by prefacing the filename with the '@' character. Each line of this file is treated as an argument to the `-s` option. You may put comments in this file. All comments must start with the '#' character. This file can be used to specify all the functions in the erver that may have services mapped to them. The `-s` option may appear several times. Note that services beginning with the '_' or '.' character are reserved for system use, and `buildserver` will fail if the `-s` option is used to include such a service in the server.

`-C`

specifies COBOL compilation. `buildserver` normally uses the `cc` command to produce the `a.out`. In order to allow for the specification of an alternate compiler, `buildserver` checks for the existence of a shell variable named `CC`. If `CC` does not exist in `buildserver's` environment, or if it is the string "", `buildserver` will use `cc` as the compiler. If `CC` does exist in the environment, its value is taken to be the name of the compiler to be executed. Likewise, the shell variable `CFLAGS` is taken to contain a set of parameters to be passed to the compiler.

`-k`

keeps the server main stub. `buildserver` generates a main stub with data structures such as the service table and a `main()` function. This is normally compiled and then removed when the server is built. This option indicates that the source file should be kept (to see what the source file name is, use the `-v` option).

Note: The generated contents of this file may change from release to release; DO NOT count on the data structures and interfaces exposed in this file. This option is provided to aid in debugging of build problems.

Environment Variables

Same as `buildclient`.

Examples

The following example shows how to specify the resource manager (`-r Tuxedo/SQL`) libraries on the `buildserver` command line:

```
buildserver -r Tuxedo/SQL -s OPEN_ACCT -s CLOSE_ACCT  -o ACCT
-f ACCT.o -f appinit.o -f util.o
```

The following example shows how `buildserver` can be supplied `CC` and `CFLAGS` variables and how `-f` can be used to supply a `-lm` option to the `CC` line to link in the math library:

```
CFLAGS=-g CC=/bin/cc  buildserver -r Tuxedo/SQL -s DEPOSIT
-s WITHDRAWAL -s INQUIRY  -o TLR -f TLR.o -f util.o -f -lm
```

The following example shows use of the `buildserver` command with no resource manager specified:

```
buildserver -s PRINTER -o PRINTER -f PRINTER.o
```

tmadmin

Function BEA Tuxedo bulletin board command interpreter

Synopsis `tmadmin [ -r ] [ -c ] [ -v ]`

Description With the commands listed below, `tmadmin` provides for inspection and modification of bulletin boards and associated entities in either a uniprocessor, multiprocessor or networked environment. The `TUXCONFIG` and `TUXOFFSET` environment variables are used to determine the location and offset where the BEA Tuxedo configuration file has been loaded. If `tmadmin` is invoked with the `-c` option, it enters configuration mode. The only valid commands are `default`, `echo`, `help`, `quit`, `verbose`, `livtoc`, `crdl`, `lidl`, `dsdl`, `indl`, and `dumptlog`. `tmadmin` may be invoked in this mode on any node, including inactive nodes. A node is considered active if `tmadmin` can join the application as an administrative process or client (via a running BBL). The `-r` option instructs `tmadmin` to enter the bulletin board as a client instead of the administrator and provides read-only access. This is useful if it is desired to leave the administrator slot unoccupied. Only one `tmadmin` process can be the administrator at a time. When the `-r` option is specified by a user other than the BEA Tuxedo administrator and security is turned on, the user will be prompted for a password. The `-v` option causes `tmadmin` to display the BEA Tuxedo version number and license number. After printing out the information, `tmadmin` exits. If the `-v` option is entered with either of the other two options, the others are ignored; only the information requested by the `-v` option is displayed. Normally, `tmadmin` may be run on any active node within an active application. If it is run on an active node that is partitioned, then commands are limited to read only access to the local bulletin board. These include `bbls, bbparms, bbstat, default, dump, dumptlog, echo, help, printclient, printnet,` printqueue, printserver, printservice, printtrans, printgroup, reconnect, quit, serverparms, serviceparms, and verbose, in addition to the configuration commands.

If the partitioned node is the backup node for the MASTER (specified as the second entry on the MASTER parameter in the RESOURCES section of the configuration file), the master command is also available to make this node the MASTER for this part of the partitioned application. If the application is inactive, tmadmin can only be run on the MASTER processor. In this mode, all of the configuration mode commands are available plus the TLOG commands (crlog, dslog, and inlog) and boot.

Once tmadmin has been invoked, commands may be entered at the prompt (">") according to the following syntax: command [arguments].

Several commonly occurring arguments can be given defaults via the default command. Commands that accept parameters set via the default command check default to see if a value has been set. If one hasn't, an error message is returned. In a networked or multiprocessor environment, a single bulletin board can be accessed by setting a default machine (the logical machine id (LMID) as listed in the MACHINES section of the UBBCONFIG file). If the default machine is set to all, all bulletin boards are accessed. If machine is set to DBBL, the distinguished bulletin board is addressed. The default machine is shown as part of the prompt, as in: MASTER> .

If the machine is not set via the default command, the DBBL is addressed (the local BBL is used in a SHM configuration). The machine value for a command can generally be obtained from the default setting (printserver is an example). A caution is required here, however, because some commands (the TLOG commands, for example) act on devices found through TUXCONFIG; a default setting of DBBL or all results in an error. There are some commands where the machine value must be provided on the command line (logstart is an example); the value does not appear as an argument to the -m option. Once set, a default remains in effect until the session is ended, unless changed by another default command. Defaults may be overridden by entering an explicit value on the command line, or unset by entering the value "*". The effect of an override lasts for a single instance of the command. Output from tmadmin commands is paginated according to the pagination command in use (see the paginate subcommand below). There are some commands that have either verbose or terse output. The verbose command can be used to set the default output level. However, each command (except boot, shutdown and config) takes a -v or -t option to turn verbose or terse output on for that command only. When output is printed in terse mode, some of the information (for example, LMID or GROUP name, service or server name) may be truncated. This is indicated by a plus sign, +, at the end of the value. The entire value may be seen by re-entering the command in verbose mode.

tmadmin Commands

Commands may be entered either by their full name or their abbreviation (as given in parentheses), followed by any appropriate arguments. Arguments appearing in square brackets, [], are optional; those in curly braces, {}, indicate a selection from mutually exclusive options. Note that command line options that do not appear in square brackets need not appear on the command line (that is, they are optional) if the corresponding default has been set via the default command. Ellipses following a group of options in curly brackets, {}..., indicate that more than one of the options may appear on the command line (at least one must appear).

`aborttrans (abort) [ -yes ] [ -g groupname ] tranindex`

If `groupname` is specified (on the command line or by default), abort the transaction associated with the specified transaction index, `tranindex,` at the specified server group. Otherwise, notify the coordinator of the transaction to abort the global transaction. If the transaction is known to be decided and the decision was to commit, `aborttrans` will fail. The index is taken from the previous execution of the `printtrans` command. To completely get rid of a transaction, `printtrans` and `aborttrans` must be executed for all groups that are participants in the transaction. This command should be used with care.

The following provides a brief description of `tmadmin` commands. For a complete description see the online documentation.

`advertise (adv) {-q qaddress [ -g groupname ][-i srvid] | -g groupname -i srvid} service[:func]`
: Create an entry in the service table for the indicated service

`bbclean (bbc) machine`
: Check the integrity of all accessers of the bulletin board residing on machine `machine`, and the DBBL as well

`bbparms (bbp)`
: Print a summary of the bulletin board's parameters, such as maximum number of servers and services.

`bbsread (bbls) machine`
: List the IPC resources for the bulletin board on machine `machine`. In SHM mode, the machine parameter is optional. Information from remote machines is not available.

`bbstats (bbs)`

Print a summary of bulletin board statistics. (See also `shmstats`)

`boot (b) [options]`

This command is identical to the `tmboot` command. See `tmboot` for an explanation of options and restrictions on use.

`broadcast (bcst) [-m machine] [-u usrname] [-c cltname] [text]`

Broadcasts an unsolicited notification message to all selected clients

```
changeload (chl) [-m machine] {-q qaddress [-g groupname][-i srvid]
| -g groupname -i srvid } -s service newload
```

Change the load associated with the specified service to newload.

```
changepriority (chp) [-m machine] {-q qaddress [-g groupname][-s
srvid] | -g groupname -i srvid } -s service newpri
```

Change the dequeuing priority associated with the specified service `newpri`.

`changetrace (chtr) [-m machine] [-g groupname] [-i srvid] newspec`

Change the runtime tracing behavior of currently executing processes to `newspec`.

```
changetrantime (chtt) [-m machine] {-q qaddress [-g groupname] -
[-s srvid] | -g groupname -i srvid } -s service newtlim
```

Change the transaction timeout value associated with the specified service to `newtlim`.

`committrans (commit) [ -yes ] -g groupname tranindex`

Commit the transaction associated with the specified transaction index `tranindex` at the specified server group.

`config (conf)`

This command is identical to the `tmconfig` command.

```
crdl -b blocks -z config -o configoffset [ -O newdefoffset ] [
newdevice]
```

Create an entry in the universal device list.

`crlog (crlg) -m machine`

Create the DTP transaction log for the named or default machine (it cannot be "DBBL" or "all").

```
default (d) [-g groupname] [-i srvid] [-m machine] [-u usrname] [-
c cltname][-q qaddress] [-s service] [-b blocks] [-o offset] [-z
config] [-a { 0|1|2 }]
```

Set the corresponding argument to be the default group name, server ID, machine, user name, client name, queue address, service name, device blocks, device offset, or UDL configuration device path (it must be an absolute pathname starting with /).

`dsdl [ -yes ] -z config [ -o offset ] dlindex`
Destroy an entry found in the universal device list

`dslog (dslg) [ -yes ] -m machine`
Destroy the DTP transaction log for the named or default machine (it cannot be "DBBL" or "all").

`dump (du) filename`
Dump the current bulletin board into the file filename.

`dumptlog (dl) -z config [ -o offset ] [ -n name ] [ -g groupname ] filename`
Dumps an ASCII version of the `TLOG` into the specified filename

`echo (e) [{off | on}]`
Echo input command lines when set to on.

`help (h) [{command | all}]`
Print help messages.

`initdl (indl) [ -yes ] -z config [ -o offset ] dlindex`
Reinitializes a device on the device list.

`inlog [ -yes ] -m machine`
Reinitialize the DTP transaction log for the named or default machine (it cannot be "DBBL" or "all

`lidl -z config [ -o offset ] [ dlindex ]`
Print the universal device list.

`livtoc -z config [ -o offset ]`
Prints information for all `VTOC` table entries.

`loadtlog -m machine filename`
Read the ASCII version of a `TLOG` from the specified filename (produced by dumptlog) into the existing `TLOG` for the named or default machine (it cannot be "DBBL" or "all").

`logstart machine`
Force a warm start for the `TLOG` information on the specified machine.

`master (m) [ -yes ]`
If run on the backup node when partitioned, the backup node takes over as the acting master node and a DBBL is booted to take over administrative processing.

`migrategroup (migg) [-cancel] group_name`
: The `migrategroup` command takes the name of a server group.

`migratemach (migm) [-cancel] machine`
: All servers running on the specified machine are migrated to their alternate location.

`paginate (page) [{off | on}]`
: Paginate output.

`passwd`
: Prompt the administrator for a new application password in an application requiring security.

`pclean (pcl) machine`
: `pclean` first forces a `bbclean` on the specified machine to restart or cleanup any servers that may require it.

`printclient (pclt) [-m machine] [-u usrname] [-c cltname]`
: Print information for the specified set of client processes.

`printconn (pc) [-m machine]`
: Print information about conversational connections.

`printgroup (pg) [-m machine] [-g groupname]`
: Print server group table information.

`printnet (pnw) [ mach_list ]`
: Print network connection information.

`printqueue (pq) [qaddress]`
: Print queue information for all application and administrative servers.

`printserver (psr) [-m machine] [-g groupname] [-i srvid] [-q qaddress]`
: Print information for application and administrative servers.

`printservice (psc) [-m machine] [-g groupname] [-i srvid] [-a { 0|1|2 }][-q qaddress] [-s service]`
: Print information for application and administrative services.

`printtrans (pt) [-g groupname] [-m machine]`
: Print global transaction table information for either the specified or the default machine.

`quit (q)`
: Terminate the session.

`reconnect (rco) non-partitioned_machine1 partitioned_machine2`

Initiate a new connection from the non-partitioned machine to the partitioned machine.

`resume (res) {-q qaddress | -g groupname | -i srvid | -s service} ...`

Resume (unsuspend) services.

`serverparms (srp) -g groupname -i srvid`

Print the parameters associated with the server specified by `groupname` and `srvid` for a group.

`serviceparms (scp) -g groupname -i srvid -s service`

Print the parameters associated with the service specified by `groupname`, `srvid` and `service`.

`shmstats (sstats) [ ex | app ]`

If `MODEL SHM` is specified in the configuration file, `shmstats` can be used to assure more accurate statistics.

`shutdown (stop) [options]`

This command is identical to the `tmshutdown` command.

`suspend (susp) {-q qaddress | -g groupname | -i srvid | -s service} ...`

Suspend services.

`unadvertise (unadv) {-q qaddress [-g groupname] [-i srvid] | -g groupname -i srvid} service`

Remove an entry in the service table for the indicated service.

`verbose (v) [{off | on}]`

Produce output in verbose mode.

`! shellcommand`

Escape to shell and execute shellcommand.

`!!`

Repeat previous shell command.

`# [text]`

Lines beginning with "#" are comment lines and are ignored.

`CR>`

Repeat the last command.

Environment Variables

`tmadmin` acts as an application client if the `-r` option is used or if it cannot register as the application administrator. If this is the case, then the `APP_PW` environment variable must be set to the application password in a security application if standard input is not from a terminal.

Diagnostics

If the `tmadmin` command is entered before the system has been booted, the following message is displayed:

```
No bulletin board exists. Entering boot mode
>
```

`tmadmin` then waits for a boot command to be entered. If the `tmadmin` command is entered, without the `-c` option, on an inactive node that is not the MASTER, the following message is displayed and the command terminates:

```
Cannot enter boot mode on non-master node.
```

If an incorrect application password is entered or is not available to a shell script through the environment, then a log message is generated, the following message is displayed and the command terminates:

```
Invalid password entered.
```

tmboot

Function

Bring up a BEA Tuxedo configuration

Synopsis

```
tmboot [-l lmid] [-g grpname] [-i srvid] [-s aout] [-o sequence]
[-S] [-A] [-b] [-B lmid] [-T grpname] [-e command] [-w] [-y] [-q]
[-n] [-c] [-M] [-d1]
```

Description

`tmboot` brings up a BEA Tuxedo application in whole or in part depending on the options specified. `tmboot` can be invoked only by the administrator of the bulletin board (as indicated by the `UID` parameter in the configuration file) or by root. `tmboot` can be invoked only on the machine identified as `MASTER` in the RESOURCES section of the configuration file, or the backup acting as the `MASTER`, that is, with the DBBL already running (via the master command in `tmadmin`). Except, if the `-b` option is used, the system can be booted from the backup machine without it having been designated as the `MASTER`. With no options, `tmboot` executes all administrative processes and all servers listed in the SERVERS section of the configuration file named by the environment variables, `TUXCONFIG` and `TUXOFFSET`. If the `MODEL` is `MP`, a DBBL administrative server is started on the machine indicated by the `MASTER` parameter in the RESOURCES section. An administrative server (BBL) is started on every machine listed in the MACHINES section. For each group in the GROUPS section, TMS servers are started based on the `TMSNAME` and `TMSCOUNT` parameters for each entry. All administrative servers are started followed by servers in the SERVERS sections. Any TMS or gateway servers for a group are booted before the first application server in the group is booted. The `TUXCONFIG` file is propagated to remote machines as necessary. `tmboot` normally waits for a booted process to complete its initialization (that is, `tpsvrinit()`) before booting the next process. Booting a gateway server implies that the gateway advertises its administrative service, and also advertises the application services representing the foreign services based on the `CLOPT` parameter for the gateway (`-A` will cause all services defined when the gateway is built with `buildgateway` to be advertised; `-s` can be used to give a list of services). If the instantiation has the concept of foreign servers, these servers are booted by the gateway at this time. Booting an LMID is equivalent to booting all groups on that LMID. Application servers are booted in the order specified by the `SEQUENCE` parameter, or in the order of server entries in the configuration file (see description in `ubbconfig`). If two or more servers in the SERVERS section of the configuration file have the same `SEQUENCE` parameter, then `tmboot` may boot these servers in parallel and will not continue until they all complete initialization. Each entry in the SERVERS section can have a `MIN` and `MAX` parameter. `tmboot` boots `MIN` application servers (the default is `1` if `MIN` is not specified for the server entry) unless the `-i` option is specified; using the `-i` option causes individual servers to be booted up to `MAX` occurrences. If a server can not be started, a diagnostic is written on the central event log (and to the standard output, unless `-q` is specified), and `tmboot` continues -- except that if the failing process is a BBL, servers that depend on that BBL are silently ignored; if the failing process is a DBBL, `tmboot` ignores the rest of the configuration file. If a server is configured with an alternate LMID and fails to start on its primary machine, `tmboot` automatically attempts to start the server on the alternate machine and, if successful, sends a message to the DBBL to update the server group section of `TUXCONFIG`. For servers in the SERVERS section, only `CLOPT`, `SEQUENCE`, `SRVGRP` and `SRVID` are used by `tmboot`. Collectively, these are known as the server's boot parameters. Once the server has been booted, it reads the configuration file to find its runtime parameters. (See `ubbconfig(5)` for a description of all parameters.) All administrative and application servers are booted with `APPDIR` as their current working directory. The value of `APPDIR` is specified in the configuration file in the MACHINES section for the machine on which the server is being booted. The search path for the server executables is `APPDIR`, followed by `TUXDIR/bin`, followed by `/bin` and `/usr/bin`, followed by any PATH specified in the `ENVFILE` for the MACHINE. The search path is only used if an absolute path name is not specified for the server. Values placed in the server's `ENVFILE` are not used for the search path. When a server is booted, the variables `TUXDIR`, `TUXCONFIG`, `TUXOFFSET`, and `APPDIR`, with values specified in the configuration file for that machine, are placed in the environment. The environment variable `LD_LIBRARY_PATH` is also placed in the environment of all servers. Its value defaults to `$APPDIR:$TUXDIR/lib:/lib:/usr/lib:lib>` where `lib>` is the value of the first `LD_LIBRARY_PATH=` line appearing in the machine `ENVFILE`. See `ubbconfig` for a description of the syntax and use of the `ENVFILE`. The `ULOGPFX` for the server is also set up at boot time based on the parameter for the machine in the configuration file. If not specified, it defaults to `$APPDIR/ULOG`. All of these operations are performed before the application initialization function, `tpsvrinit()`, is called. Many of the command line options of `tmboot` serve to limit the way in which the system is booted and can be used to boot a partial system. The following options are supported:

`-l lmid`

For each group whose associated `LMID` parameter is `lmid`, all TMS and gateway servers associatedmwith the group are booted and all servers in the SERVERS section associated with those groups are executed.

`-g grpname`

All TMS and gateway servers for the group whose `SRVGRP` parameter is `grpname` are started followed by all servers in the SERVERS section associated with that group. TMS servers are started based on the `TMSNAME` and `TMSCOUNT` parameters for the group entry.

`-i srvid`

All servers in the SERVERS section whose `SRVID` parameter is `srvid` are executed.

`-s aout`

All servers in the SERVERS section with name `aout` are executed. This option can also be used to boot TMS and gateway servers; normally this option would be used in this way in conjunction with the `-g` option.

`-o sequence`

All servers in the SERVERS section with `SEQUENCE` parameter `sequence` are executed.

`-S`

All servers in the SERVERS section are executed.

`-A`

All administrative servers for machines in the MACHINES section are executed. Use this option to guarantee that the DBBL and all BBL and

BRIDGE processes are brought up in the correct order (also see the `-M` option).

`-b`

Boot the system from the BACKUP machine, (without having to make it the MASTER).

`-B lmid`

A BBL is started on a processor with logical name lmid.

`-M`

This option starts administrative servers on the master machine. If the MODEL is `MP`, a DBBL administrative server is started on the machine indicated by the `MASTER` parameter in the RESOURCES section. A BBL is started on the MASTER machine, and a `BRIDGE` is started if the `LAN` option and a `NETWORK` entry are specified in the configuration file.

`-d1`

Causes command line options to be printed on the standard output. Useful when preparing to use `sdb` to debug application services.

`-T grpname`

All TMS servers for the group whose `SRVGRP` parameter is `grpname` are started (based on the `TMSNAME` and `TMSCOUNT` parameters associated with the group entry). This option is the same as booting based on the TMS server name (`-s` option) and the group name (`-g`).

`-e command`

Causes command to be executed if any process fails to boot successfully. command can be any program, script, or sequence of commands understood by the command interpreter specified in the `SHELL` environment variable. This allows an opportunity to bail out of the boot procedure. If command contains white space, the entire string must be enclosed in quotes. This command is executed on the machine on which `tmboot` is being run, not on the machine where the server is being booted.

`-w`

Informs `tmboot` not to wait for servers to complete initialization before booting another server. This option should be used with caution. BBLs depend on the presence of a valid DBBL, ordinary servers require a running BBL on the processor on which they are placed. These conditions can not be guaranteed if servers are not started in a synchronized manner.This option overrides the waiting that is normally done when servers have sequence numbers.

`-y`

Assumes a yes answer to a prompt that asks if all administrative and server processes should be booted. (The prompt appears only when the command is entered with none of the limiting options.)

`-q`

Suppresses the printing of the execution sequence on the standard output. It implies `-y`.

`-n`

The execution sequence is printed, but not performed.

`-c`

Minimum IPC resources needed for this configuration are printed. When the `-l, -g, -i, -o,` and `-s` options are used in combination, only servers that satisfy all qualifications specified will be booted. The `-l, -g, -s,` and `-T` options cause TMS servers to be booted; the `-l, -g,` and `-s` options cause gateway servers to be booted; the `-l, -g, -i, -o, -s,` and `-S` options apply to application servers. Options that boot application servers will fail if a BBL is not available on the machine.The `-A, -M,` and `-B` options apply only to administrative processes. The standard input, standard output, and standard error file descriptors will be closed for all booted servers.

Environment Variables

During the installation process, an administrative password file is created. When necessary, BEA Tuxedo searches for this file in the following directories (in the order shown): `APPDIR/.adm/tlisten.pw TUXDIR/udataobj/tlisten.pw` To ensure that your password file will be found, make sure you have set the `APPDIR` and/or `TUXDIR` environment variables.

Diagnostics

If `TUXCONFIG` is set to a non-existent file, two fatal error messages are displayed: error processing configuration file configuration file not found If `tmboot` fails to boot a server, it will exit with exit code `1` and the user log should be examined for further details; otherwise it will exit with exit code `0`. If `tmboot` is run on an inactive non-master node, a fatal error message is displayed: `tmboot` cannot run on a non-master node. If `tmboot` is run on an active node that is not the acting master node, a fatal error message is displayed:

```
tmboot cannot run on a non acting-master node in an active
application.
```

If the same `IPCKEY` is used in more than one `TUXCONFIG` file, `tmboot` fails with the following message:

```
Configuration file parameter has been changed since last tmboot
```

If there are multiple node names in the MACHINES section in a non-LAN configuration, a fatal error message is displayed: Multiple nodes not allowed in MACHINES for non-LAN application.

Examples

To start only those servers located on the machines logically named CS0 and CS1:

```
tmboot -l CS0 -l CS1
```

To start only those servers named CREDEB and belonging to group DBG1:

```
tmboot -g DBG1 -s CREDEB1
```

To boot a BBL on the machine logically named PE8, as well as all those servers whose location is specified as PE8:

```
tmboot -B PE8 -l PE8
```

To view minimum IPC resources needed for the configuration:

```
tmboot -c
```

tmloadcf

Function

Parse a `UBBCONFIG` file and load binary `TUXCONFIG` configuration file

Synopsis

```
tmloadcf [-n] [-y] [-c] [-b blocks] {ubbconfig_file | -}
```

Description

`tmloadcf` reads a file or the standard input that is in `UBBCONFIG` syntax, checks the syntax, and optionally loads a binary `TUXCONFIG` configuration file. The `TUXCONFIG` and (optionally) `TUXOFFSET` environment variables point to the `TUXCONFIG` file and (optional) offset where the information should be stored. `tmloadcf` can only be run on the MASTER machine, as defined in the RESOURCES section of the `UBBCONFIG` file, unless the `-c` or `-n` option is specified. `tmloadcf` prints a warning message if it finds any section of the `UBBCONFIG` file missing, other than a missing NETWORK section in a configuration where the `LAN OPTION` is not specified (see `ubbconfig`) or a missing ROUTING section. If a syntax error is found while parsing the input file, `tmloadcf` exits without performing any updates to the `TUXCONFIG` file. The effective user identifier of the person running `tmloadcf` must match the `UID`, if specified, in the RESOURCES section of the `UBBCONFIG` file. The `-c` option to `tmloadcf` causes the program to print minimum IPC resources needed for this configuration. Resource requirements that vary on a per-processor basis are printed for each processor in the configuration. The TUXCONFIG file is not updated. The -n option to tmloadcf causes the program to do only syntax checking of the ASCII UBBCONFIG file without actually updating the TUXCONFIG file. After syntax checking, tmloadcf checks to see if the file pointed to by TUXCONFIG exists, is a valid BEA Tuxedo system file system, and contains TUXCONFIG tables. If these conditions are not true, the user is prompted to decide if they want tmloadcf to create and initialize the file with Initialize TUXCONFIG file: path [y, q]? Prompting is suppressed if the standard input or output are not terminals, or if the -y option is specified on the command line. Any response other than "y" or "Y" will cause tmloadcf to exit without creating the configuration file. If the TUXCONFIG file is not properly initialized, and the user has given the go-ahead, tmloadcf creates the BEA Tuxedo system file system and then creates the TUXCONFIG tables. If the -b option is specified on the command line, its argument is used as the number of blocks for the device when creating the BEA Tuxedo system file system. If the value of the -b option is large enough to hold the new TUXCONFIG tables, tmloadcf will use the specified value to create the new file system; otherwise, tmloadcf will print an error message and exit. If the -b option is not specified, tmloadcf will create a new file system large enough to hold the TUXCONFIG tables. The -b option is ignored if the file system already exists. The -b option is highly recommended if TUXCONFIG is a raw device (that has not been initialized) and should be set to the number of blocks on the raw device. The -b option is not recommended if TUXCONFIG is a regular UNIX file. If the TUXCONFIG file is determined to already have been initialized, tmloadcf ensures that the system described by that TUXCONFIG file is not running. If the system is running, tmloadcf prints an error message and exits. If the system is not running and TUXCONFIG file already exists, tmloadcf will prompt the user to confirm that the file should be overwritten with Really overwrite TUXCONFIG file [y, q]? Prompting is suppressed if the standard input or output are not a terminal or if the -y option is specified on the command line. Any response other than "y" or "Y" will cause tmloadcf to exit without overwriting the file. If the SECURITY parameter is specified in the RESOURCES section of the configuration, then tmloadcf will flush the standard input, turn off terminal echo and prompt the user for an application password as follows:

```
Enter Application Password?
Reenter Application Password?
```

The password is limited to 30 characters. The option to load the ASCII UBBCONFIG file via the standard input (rather than a file) cannot be used when the SECURITY parameter is turned on. If the standard input is not a terminal, that is, if the user cannot be prompted for a password (as with a here file, for example), then the environment variable APP_PW is accessed to set the application password. If the environment variable APP_PW is not set with the standard input not a terminal, then tmloadcf will print an error message, generate a log message and fail to load the TUXCONFIG file. Assuming no errors, and if all checks have passed, tmloadcf loads the UBBCONFIG file into the TUXCONFIG file. It will overwrite all existing information found in the TUXCONFIG tables. Note that some values are rounded during the load and may not match when they are unloaded. These include but are not limited to MAXRFT and MAXRTDATA.

Environment Variables

The environment variable APP_PW must be set for applications that have the SECURITY parameter is specified and run tmloadcf with something other than a terminal as the standard input.

Examples

To load a configuration file from UBBCONFIG file BB.shm, initialized the device with 2000 blocks:

```
tmloadcf -b2000 -y BB.shm
```

Diagnostics

If an error is detected in the input, the offending line is printed to standard error along with a message indicating the problem. If a syntax error is found in the UBBCONFIG file or the system is currently running, no information is updated in the TUXCONFIG file and tmloadcf exits with exit code 1. If tmloadcf is run by a person whose effective user identifier doesn't match the UID specified in the UBBCONFIG file, the following error message is displayed:

```
*** UID is not effective user ID ***
```

If tmloadcf is run on a non-master node, the following error message is displayed:

```
tmloadcf cannot run on a non-master node.
```

If tmloadcf is run on an active node, the following error message is displayed:

```
tmloadcf cannot run on an active node.
```

Upon successful completion, tmloadcf exits with exit code 0. If the TUXCONFIG file is updated, a userlog message is generated to record this event.

tmshutdown

Function

Shutdown a set of BEA Tuxedo servers

Synopsis

```
tmshutdown [options]
```

Description `tmshutdown` stops the execution of a set of servers or removes the advertisements of a set of services listed in a configuration file. Only the administrator of the bulletin board (as indicated by the `UID` parameter in the configuration file) or root can invoke the `tmshutdown` command. `tmshutdown` can be invoked only on the machine identified as `MASTER` in the RESOURCES section of the configuration file, or the backup acting as the `MASTER`, that is, with the DBBL already running (via the master command in `tmadmin`). An exception to this is the `-P` option which is used on partitioned processors (see below). With no options, `tmshutdown` stops all administrative, TMS, and gateway servers, and servers listed in the SERVERS section of the configuration file named by the `TUXCONFIG` environment variable and removes their associated IPC resources. For each group, all servers in the SERVERS section, if any, are shutdown followed by any associated gateway servers (for foreign groups) and TMS servers. Administrative servers are shutdown last. Application servers without `SEQUENCE` parameters are shutdown first in reverse order of the server entries in the configuration file, followed by servers with `SEQUENCE` parameters that are shutdown from high to low sequence number. If two or more servers in the SERVERS Section of the configuration file have the same `SEQUENCE` parameter, then `tmshutdown` may shut down these servers in parallel. Each entry in the SERVERS Section may have an optional `MIN` and `MAX` parameter. `tmshutdown` shuts down all occurrences of a server (up to `MAX` occurrences) for each server entry, unless the `-i` option is specified; using the `-i` option causes individual occurrences to be shut down. If it is not possible to shutdown a server, or remove a service advertisement, a diagnostic is written on the central event log (see `userlog`). The following is a description of all options:

`-l lmid`

For each group whose associated `LMID` parameter is `lmid`, all servers in the SERVERS section associated with the group are shut down, followed by any TMS and gateway servers associated with the group.

`-g grpname`

All servers in the SERVERS section associated with the specified group (that is, whose `SRVGRP` parameter is `grpname`) are shutdown, followed by all TMS and gateway servers for the group. TMS servers are shutdown based on the `TMSNAME` and `TMSCOUNT` parameters for the group entry. For a foreign group, the gateway servers for the associated entry in the HOST section are shutdown based on `GATENAME` and `GATECOUNT`. Shutting down a gateway implies its administrative service and all advertised foreign services are unadvertised, in addition to stopping the process.

`-i srvid`

All servers in the SERVERS section whose `SRVID` parameter is `srvid` are shutdown. Do not enter a `SRVID` greater than 30,000; this indicates system processes (that is, TMSs or gateway servers) that should only be shutdown via the `-l` or `-g` options.

`-s aout`
: All servers in the SERVERS section with name `aout` are shutdown. This option can also be used to shutdown TMS and gateway servers.

`-o sequence`
: All servers in the SERVERS section with `SEQUENCE` parameter sequence are shutdown.

`-S`
: All servers in the SERVERS section are shutdown.

`-A`
: All administrative servers are shutdown.

`-M`
: This option shuts down administrative servers on the master machine. The BBL is shut down on the MASTER machine, and the BRIDGE is shut down if the LAN option and a NETWORK entry are specified in the configuration file. If the MODEL is `MP`, the DBBL administrative server is shut down.

`-B lmid`
: The BBL on the processor with logical name lmid is shutdown.

`-T grpname`
: All TMS servers for the server group whose `SRVGRP` parameter is `grpname` are shut down (based on the `TMSNAME` and `TMSCOUNT` parameters associated with the server group entry).

`-w delay`
: Tells `tmshutdown` to suspend all selected servers immediately and waits for shutdown confirmation for only delay seconds before forcing the server to shut down by sending a `SIGTERM` and then a `SIGKILL` signal to the server.
Note: Servers to which the `-w` option may be applied should not catch the UNIX signal `SIGTERM`.

`-k {TERM|KILL}`
: `tmshutdown` suspends all selected servers immediately and forces them to shut down in an orderly fashion (`TERM`) or preemptively (`KILL`).
Note: This option maps to the UNIX signals `SIGTERM` and `SIGKILL` on platforms which support them. By default, a `SIGTERM` initiates orderly

shutdown in a BEA Tuxedo server. Application resetting of `SIGTERM` could cause to be unable to shutdown the server.

`-y`
: Assumes a yes answer to a prompt that asks if all administrative and server processes should be shutdown. (The prompt appears only when the command is entered with none of the limiting options.)

`-q`
: Suppresses the printing of the execution sequence on the standard output. It implies `-y`.

`-n`
: The execution sequence is printed, but not performed.

`-R`
: For migration operations only, shuts down a server on the original processor without deleting its bulletin board entry in preparation for migration to another processor. The `-R` option must be used with either the `-l` or `-g` option (e.g., `tmshutdown -l lmid -R`) The `MIGRATE` option must be specified in the RESOURCES section of the configuration file.

`-c`
: Shuts down BBLs even if clients are still attached.

`-H lmid`
: On a uniprocessor, all administrative and applications servers on the node associated with the specified `lmid` are shut down. On a multiprocessor(e.g., 3B4000), all PEs are shut down, even if only one PE is specified.

`-P lmid`
: With this option, `tmshutdown` attaches to the bulletin board on the specified `lmid`, ensures that this `lmid` is partitioned from the rest of the application (that is, does not have access to the DBBL), and shuts down all administrative and application servers. It must be run on the processor associated with the `lmid` in the MACHINES section of the configuration file. The `-l, -g, -s,` and `-T` options cause TMS servers to be shut down; the `-l, -g,` and `-s` options cause gateway servers to be shut down; the `-l, -g, -i, -s, -o,` and `-S` options apply to application servers; the `-A, -M,` and `-B` options apply only to administrative processes. When the `-l, -g, -i, -o,` and `-s` options are used in combination, only servers that satisfy all qualifications specified will be shut down. If the distributed transaction processing feature is being used such that global transactions are in progress when servers are shutdown, transactions that have not yet reached the point where commit is logged after

pre-commit will be aborted; transactions that have reached the commit point will be completed when the servers (for example, TMS) are booted again.

Diagnostics

If `tmshutdown` fails to shut down a server or a fatal error occurs, it will exit with exit code `1` and the user log should be examined for further details; otherwise it will exit with exit code `0`. If `tmshutdown` is run on an active node that is not the acting master node, a fatal error message is displayed:

```
tmshutdown cannot run on a non acting-master node in an active
application.
```

If shutting down a process would partition active processes from the DBBL, a fatal error message is displayed:

```
cannot shutdown, causes partitioning.
```

If a server has died, the following somewhat ambiguous message is produced:
`CMDTUX_CAT:947`

```
Cannot shutdown server GRPID
```

Examples

To shutdown the entire system and remove all BEA Tuxedo IPC resources (force it if confirmation not received in 30 seconds):

```
tmshutdown -w 30
```

To shutdown only those servers located on the machine with `lmid` of CS1. Since the `-l` option restricts the action to servers listed in the SERVERS section, theBBL on CS1 is not shutdown:

```
tmshutdown -l CS1
```

ud, ud32, wud, wud32

Function

BEA Tuxedo driver program

Synopsis

```
ud [-p] [-ddelay] [-eerror_limit] [-r] [-ssleeptime] [-ttimeout]
[-n]  [-u {n | u | j}]  [-Uusrname]  [-Ccltname] [-Sbuffersize]
ud32 [options]
wud [options]
wud32 [options]
```

Description

ud reads an input packet from its standard input using Fextread. The packet must contain a field identified as the name of a service. The input packet is transferred to an FML fielded buffer (FBFR) and sent to the service. If the service that receives the FBFR is one that adds records to a database, ud provides a method for entering bulk fielded data into a database known to the BEA Tuxedo system. By using flags (see INPUT FORMAT) to begin the lines of the input packet, you can use ud to test BEA Tuxedo services. By default, after sending the FBFR to the service, ud expects a return FBFR. The sent and reply FBFRs are printed to ud's standard output; error messages are printed to standard error. ud32 uses FML32 buffers of type FBFR32. wud and wud32 are versions of ud and ud32 built using the Workstation libraries. On sites supporting just Workstation, only the wud and wud32 commands will be present.

Options

ud supports the following options:

-p
: suppress printing of the sent and returned fielded buffers.

-d
: expect a delayed reply for every request. delay specifies the maximum delay time in seconds before time out. If time-out occurs, an error message is printed on stderr. If ud receives reply messages for previous requests within the delay time, they will be indicated as delayed RTN packets. Hence, it is possible to receive more than one reply packet within a delay time interval. The -d option is not available for wud on DOS operating systems.

-e error_limit
: ud stops processing requests when errors exceed the limit specified in error_limit. If no limit is specified, the default is 25.

-r
: ud should not expect a reply message from servers.

-s sleeptime
: sleep between sends of input buffers. sleeptime is the time, in seconds, of the sleep.

-t timeout
: ud should send requests in transaction mode. timeout is the time, in seconds, before the transaction is timed out. The -d delay and -r (no reply) options are not allowed in combination with the -t option.

-u {n | u | j}
: specify how the request buffer is modified before reading each new packet. The n option indicates that the buffer should be reinitialized (treated as new). The u option indicates that the buffer should be updated with the reply buffer

using `Fupdate`. The `j` option indicates that the reply buffer should be joined with the request buffer using `Fojoin`.

`-n`
: reinitialize the buffer before reading each packet (i.e., treat each buffer as a new buffer). This option is equivalent to `-un` and is maintained for compatibility.

`-U usrname`
: Use `usrname` as the user name when joining the application.

`-S buffersize`
: If the default buffer size is not large enough, the `-S` option can be used to raise the limit. `buffersize` can be any number up to `MAXLONG`. The `-d` delay and `-r` options are mutually exclusive.

Input Format

Input packets consist of lines formatted as follows:

`[flag]fldname fldval`

`flag` is optional. If flag is not specified, a new occurrence of the field named by `fldname` with value `fldval` is added to the fielded buffer. If flag is specified, it should be one of:

`+`
: occurrence `0` of `fldname` in `FBFR` should be changed to `fldval`.

`-`
: occurrence `0` of `fldname` should be deleted from `FBFR`. The tab character is required; `fldval` is ignored.

`=`
: the value in `fldname` should be changed. In this case, `fldval` specifies the name of a field whose value should be assigned to the field named by `fldname`.

`#`
: the line is treated as a comment and is ignored.

If `fldname` is the literal value `SRVCNM`, `fldval` is the name of the service to which `FBFR` is to be passed. Lengthy field values can be continued on the next line by having the continuation line begin with a tab. A line consisting only of the newline character ends the input and sends the packet to `ud`. If an input packet begins with a line consisting of the character `n` followed by the newline character, the `FBFR` is reinitialized. `FBFR` reinitialization can be specified for all packets with the `-un` option on the command line. To enter an unprintable character in the input packet, use the escaping convention followed by the hexadecimal representation of the desired character (see `ASCII(5)` in a UNIX reference manual). An additional backslash is needed to protect the escape from the shell. A space, for example, can be entered in the input data as 20. `ud` recognizes all input in this format, but its greatest usefulness is for non-printing characters.

Processing Model

Initially, `ud` reads a fielded buffer from its standard input and sends it to the service whose name is given by the `fldval` of the line where `fldname` equals `SRVCNM`. Unless the `-r` option is selected, `ud` waits for a reply fielded buffer. After obtaining the reply, `ud` reads another fielded buffer from the standard input. In so doing, `ud` retains the returned buffer as the current buffer. This means that the lines on the standard input that form the second fielded buffer are taken to be additions to the buffer just returned. That is, the default action is for `ud` to maintain a current buffer whose contents are added to by a set of input lines. The set is delimited by a blank line. `ud` may be instructed to discard the current buffer (that is, to reinitialize its `FBFR` structure) either by specifying the `-un` option on the command line, or by including a line whose only character is the letter `n` as the first line of an input set. `ud` may be instructed to merge the contents of the reply buffer into the request buffer by specifying either the `-uu` option (`Fupdate` is used) or the `-uj` option (`Fojoin` is used).

Environment Variables

`FLDTBLDIR` and `FIELDTBLS` must be set and exported. `FLDTBLDIR` must include `$TUXDIR/udataobj` in the list of directories. `FIELDTBLS` must include `Usysflds` as one of the field tables.

`APP_PW` must be set to the application password in a security application if standard input is not from a terminal. `TPIDATA` must be set to the application specific data necessary to join the application in a security application with an authentication server if standard input is not from a terminal.

`WSNADDR`, `WSDEVICE` and optionally `WSTYPE` must be set if access is from a workstation. See compilation for more details on setting environment variables for client processes.

Diagnostics

`ud` fails if it cannot become a client process, if it cannot create the needed `FBFR`s, or if it encounters a UNIX system error. It also fails if it encounters more than 25 errors in processing a stream of input packets. These can be syntax errors, missing service names, errors in starting or committing a transaction, time-outs and errors in sending the input `FBFR` or in receiving the reply `FBFR`.

Examples

```
$ud <EOF>
SRVCNM BUY
CLIENT J. Jones
ADDR 21 Valley Road
```

```
STOCK AAA
SHARES 100
<CR>
+SRVCNM SELL
+STOCK XXX
+SHARES 300
STOCK YYY
SHARES 150
<CR>
n
SRVCNM BUY
CLIENT T. Smith
ADDR 1 Main Street
STOCK BBB
SHARES 175
<CR>
+SRVCNM SELL
+STOCK ZZZ
+SHARES 100
<CR>
EOF
$
```

In this example, `ud` first sends a fielded buffer to the service `BUY` with `CLIENT` field set to J. Jones, `ADDR` field set to 21 Valley Road, `STOCK` field to AAA, and `SHARES` field set to 100. When the fielded buffer is returned from the `BUY` service, `ud` uses the next set of lines to change `SRVCNM` to SELL, `STOCK` to XXX, and `SHARES` to 300. Also, it creates an additional occurrence of the `STOCK` field with value YYY and an additional occurrence of the `SHARES` field with value 150. This fielded buffer is then sent to the `SELL` service (the new value of the `SRVCNM` field). When SELL sends back a reply fielded buffer, `ud` discards it by beginning the next set of lines with a line containing only the character n. `ud` then begins building an entirely new input packet with a `SRVCNM` of BUY, `CLIENT` of value T. Smith, and so on.

E Servopts

The following information about Servopts is excerpted from the Tuxedo Online Documentation. For additional details and a complete list of Tuxedo functions and commands, see http://edocs.beasys.com/tuxedo/tux65/index.htm.

Synopsis

```
AOUT CLOPT= [-A][-s{@filename|service[,service...][:func]}]
[-e stderr_file][-p [L][low_water][,[terminate_time]]
[:[high_water][,create_time]][-h][-l locktype][-n prio]
[-o stdout_file][-r][ -- uargs]
```

Description

`servopts` is not a command. Rather, it is a list of run-time options recognized by servers in a BEA

Tuxedo system. The server using these options may be one of the BEA Tuxedo system-supplied servers such as `FRMPRT`, or it may be an application-supplied server built with the `buildserver` command. Running servers in a BEA Tuxedo system is accomplished through the `tmboot` and `tmadmin` commands working with servers (and other resources) specified in the application configuration file. Desired selections from the `servopts` list are specified with the server in the configuration file. The following options are recognized:

`-A`
: indicates that the server should initially offer all services with which it was constructed. For BEA Tuxedo system-supplied servers, `-A` is the only way of specifying services.

`-s { @filename | service[,service...][:func] }`
: specifies the names of services to be advertised when the server is booted. In the most common case, a service is performed by a function that carries the same name; that is, the x service is performed by function x. For example, the specification

`-s x,y,z`
: will run the associated server initially offering services x, y, and z, each processed by a function of the same name. In other cases, a service (or several services) may be performed by a function of a different name. The

specification, `s x,y,z:abc` runs the associated server with initial services x, y, and z, each processed by the function abc. Spaces are not allowed between commas. Function name is preceded by a colon. Service name (and implicit function names) must be less than or equal to 15 characters in length. An explicit function name (that is, a name specified after a colon) can be up to 128 characters in length. Names longer than these limits are truncated with a warning message. When retrieved by `tmadmin` or `TM_MIB`, only the first 15 characters of a name are displayed. A filename can be specified with the `-s` option by prefacing the filename with the '@' character. Each line of this file is treated as an argument to the `-s` option. You may put comments in this file. All comments start with '#' or ':'. The -s option may be specified multiple times.

`-e`

specifies the name of a file to be opened as the server's standard error file. Providing this option ensures that a restarted server has the same standard error file as its predecessors. If this option is not used, a default diversion file called `stderr` is created in the directory specified by `$APPDIR`.

`-p [L][low_water][,[terminate_time]][:[high_water][,create_time]]`

This option can be used to support automatic spawning/decay of servers. It may be used for servers on an MSSQ with `MAX` greater than 1; it is not allowed (and not necessary) for conversational servers. Arguments to the option have the following meanings: `L` The decision to spawn more servers is based on load rather than number of servers or messages. -- the remaining arguments, `low_water`, `terminate_time`, `high_water`, and `create_time` are used to control when servers are spawned or deactivated. The algorithm is: if the load meets or exceeds `high_water` for at least `create_time` seconds, a new server is spawned. If the load drops below `low_water` for at least `terminate_time` seconds, a server is deactivated. The `L` option works only in `SHM` mode with load balancing turned on. If `SHM/LDBAL+Y` is not set, then a userlog message (`LIBTUX_CAT:1542`) is printed and no spawning is done. `low_water` defaults to an average of 1 server or message on the MSSQ or a workload of 50. `high_water` defaults to an average of 2 servers or messages, or a workload of 100. `create_time` defaults to 50: `terminate_time` defaults to 60.

`-h`

do not run the server immune to hangups. If not supplied, the server ignores the hangup signal.

`-l locktype`

lock the server in core. The argument for locktype is `t`, `d`, or `p` according to whether the text (`TXTLOCK`), data (`DATLOCK`), or the entire process (text and data - `PROCLOCK`), should be locked. See `plock` for details. The lock fails if the server is not run as root. There is no way to unlock a server once it is locked.

`-n prio`

`nice` the server according to the `prio` argument. Giving the process better priority (a negative argument) requires it to be run with the `uid` of root. See `nice(2)` for details.

`-o stdout_file`

specifies the name of a file to be opened as the server's standard output file. Providing this option ensures that a restarted server has the same standard output file as its predecessors. If this option is not used, a default diversion file called `stdout` is created in the directory specified by `$APPDIR`.

`-r`

specifies that the server should record, on its standard error file, a log of services performed. This log may be analyzed by the `txrpt(1)` command. When the `-r` option is used, make sure that the `ULOGDEBUG` variable is not set to "`y`". The `ULOGDEBUG` variable prevents debugging message from being sent to `stderr`. Debugging messages in the file will be misinterpreted by `txrpt`.

`--`

marks the end of system-recognized arguments and the start of arguments to be passed to a subroutine within the server. This option is needed only if the user wishes to supply application-specific arguments to the server. The system-recognized options precede the --; application arguments should follow it. Application arguments may be processed by a user-supplied version of the `tpsvrinit` function. `getopt(3)` should be used to parse them. Because all system arguments are processed prior to the call to `tpsvrinit(3c)`, when the call is made the external integer, `optind` points to the start of the user flags. The same option letters (for example, `-A`) may be reused after the -- argument, and given any meaning appropriate to the application.

Note: At run time the BEA Tuxedo system automatically adds the following option to each command line for each server: `-c dom=domainid`

The `-c` option adds a comment line, in which the specified domain ID is reported, to any command output that reports on the processes associated with the domain in question, such as the output of the ps command. This comment helps an administrator who is managing multiple domains to interpret a single output stream that refers to several domains.

F Error Messages

This section contains the following descriptions of error, informational, and warning messages that can be encountered while using the BEA eLink Adapter Development Kit.

Source Module adklog.c

"Unable to open <filename> for read"	
DESCRIPTION	Unable to open catalog file in `ela_parseCatFiletoBuffer()`.
ACTION	Make sure file exists, and permissions allow read access.
"malloc error"	
DESCRIPTION	Unable to allocate enough space to read catalog file in `ela_parseCatFiletoBuffer()`.
ACTION	Verify that there is sufficient memory on the machine.

Source Module cfgfns.c

"Unable to open <filename> for read"	
DESCRIPTION	Unable to open configuration file in `ela_parseFiletoBuffer()`.
ACTION	Make sure file exists, and permissions allow read access.
"malloc error"	
DESCRIPTION	Unable to allocate enough space to read catalog file in `ela_parseFiletoBuffer()`.
ACTION	Verify that there is sufficient memory on the machine.
"Buffer not large enough for file name"	
DESCRIPTION	Buffer passed to `eLA_GetConfigFileName()` not large enough to hold configuration file name specified in the `UBB` file `CLOPT` line.
ACTION	Specify a larger buffer.
"File Name parameter not found"	
DESCRIPTION	Unable to locate the `-C` filename parameter in the `UBB` config file `CLOPT` line.
ACTION	Verify that the `UBB` config file is correct.

Source Module chkelinklic.c

All messages in `eLA_chkeLinkLic()`.

"ERROR: TUXDIR is not set"	
DESCRIPTION	`TUXDIR` environment variable has not been defined.
ACTION	Define `TUXDIR`.
"ERROR: Failure reading license file <filename>"	
DESCRIPTION	This message results from a failure in `_gpdmvfile_New()`.
ACTION	Insure that the lic.txt file exists in `TUXDIR\udataobj`.
"ERROR: <platform> platform license has expired"	
DESCRIPTION	The license for the platform in question has expired.
ACTION	Obtain an up-to-date license for the platform.
"ERROR: Unlicensed platform <platform>"	
DESCRIPTION	This could result from a missing or corrupted license for the platform in question.
ACTION	Obtain a valid, up-to-date license for the platform.
"ERROR: <adapter> adapter license has expired"	
DESCRIPTION	The license for the adapter in question has expired.
ACTION	Obtain an up-to-date license for the adapter.
"ERROR: Unlicensed adapter <adapter>"	
DESCRIPTION	This could result from a missing or corrupted license for the adapter in question.
ACTION	Obtain a valid, up-to-date license for the adapter.
"ERROR: Invalid license file <filename>"	
DESCRIPTION	Signals that the version number for the adapter is incorrect.
ACTION	Obtain a valid, up-to-date license for the version number of the adapter.

"INFO: Incorrect VERSION value in <adapter> section"	
DESCRIPTION	Signals that the version number for the adapter is incorrect.
ACTION	Obtain a valid, up-to-date license for the version number of the adapter.

Glossary

ATMI

Application to Transaction Monitor Interface. The eLink Platform communications application programming interface. This is a collection of runtimes services that can be called directly by a C (or COBOL) application. These runtime services provide support for communications, distributed transactions, and system management. See the section.

BBL

The "Application" name server of the Tuxedo system. The BBL is the Tuxedo process that is aware of all servers and advertised services of a Tuxedo system. The BBL is the "name server" that connects clients to servers.

Buildserver

Tuxedo command that constructs a BEA Tuxedo server load module.

Buildclient

Tuxedo command that constructs a BEA Tuxedo client module.

Business Logic

A workflow or procedure that defines the way a company conducts business. In the eLink system business logic is automated via the business process option.

Business Process Options (BPO)

The Business Process Options is a Tuxedo service that performs workflow management.

CLOPT

Optional boot parameter in the SERVER section of the UBBCONFIG file. The value specifies the servopts that are passed to the server when the server is booted. See servopts.

Data Integration Option (DIO)

The Data Integration Option is a Tuxedo service that can translate data between different types and formats. For example COBOL copy books to FML.

FML32

The 32 bit version of Field Manipulation Language or FML. FML is a BEA proprietary data structure and function library that allows associative access to fields of a data record. The internal implementations of the record are not accessible to the users of FML. See section 2.2.

MIB

A set of classes of objects with attributes within an application. Each item in a class has particular values for the attributes.

Server

A software module that accepts requests from clients and other servers. A server advertises one or more services.

Service

An application routine available for requests by a client in the system with well-defined inputs, outputs, and processing.

Service Advertisement

The process of indicating to all participants in an application that a service is active. (This should be done dynamically by the application adapter.)

SERVOPTS

A list of run-time options recognized by servers in a BEA Tuxedo system. For complete details see Appendix G.

TUXCONFIG

The binary version of the UBBCONFIG file. It serves as the persistent part of the MIB.

Tuxedo

BEA Systems' portable Transaction Process (TP) monitor. At this point Tuxedo is synonymous with the eLink Platform, but the eLink Platform will be broadened.

UBBCONFIG

The generic Tuxedo name for the ASCII file containing the Tuxedo application configuration. This file is compiled by using the Tuxedo command tmloadcf. The resulting binary is called TUXCONFIG. (Ubbconfig files delivered as samples should be prefixed by the adapter to distinguish them from other UBBCONFIG files.)

Example: A UBBCONFIG file for the eLink FML to XML adapter (abbreviated efx) would be named:

```
efx.ubbconfig
```