TriTeal Enterprise Desktop (TED™) 4.0

Tooltalk Messaging Overview

TriTeal™

Printed in USA

Part No. TED014004000

Printing History

August 1995 ... First Edition
Printed in the U.S.A

Sales Information and Technical Support

TriTeal Corporation
2011 Palomar Airport Road, Suite 200
Carlsbad, CA 92009
USA

Tel: (619) 930-2077
Fax: (619) 930-2081
Support E-Mail: support@triteal.com
Sales E-Mail: info@triteal.com
World Wide Web: http://www.triteal.com/

Table of Contents

Preface

This book describes the TriTeal Enterprise Desktop (TED) components, commands, and error messages of the ToolTalk® service.

Note – In-depth information about the functionality of the ToolTalk service in general is beyond the scope of this book. That is, *TED ToolTalk Messaging Overview* does **not** describe ToolTalk APIs or commands, or other ToolTalk functionality not specifically related to this release of the ToolTalk service for the Common Desktop Environment. See the *Programmer's Guide* for this information.

Who Should Use This Book

This manual is for developers who create or maintain applications that use the ToolTalk service to inter-operate with other applications in Common Desktop Environment. This manual assumes familiarity with the ToolTalk service and its functionality, UNIX™ operating system commands, system administrator commands, and system terminology.

How This Book Is Organized

This book is organized as follows:

Chapter 1, "Introducing the ToolTalk Service," describes how the ToolTalk service works and how it uses information that your application supplies to deliver messages; how applications use the ToolTalk service; and application and ToolTalk components.

Chapter 2, "How to Use ToolTalk Messaging," contains the information you need to write an application using the ToolTalk service in the Common Desktop Environment, including the kinds of ToolTalk toolkit messages that need to be included in your application in order for it to inter-operate with other ToolTalk-aware Common Desktop Environment-compliant applications.

Chapter 3, "Using TTSnoop to Debug Messages and Patterns," describes how to create and send custom-constructed ToolTalk messages, and also how to selectively monitor any or all ToolTalk messages.

Chapter 4, "Using ToolTalk Tracing," describes how a ToolTalk pattern matches and delivers every message ttsession sees.

Appendix A, "The Messaging Toolkit," describes some of the application program interface (API functions) that are a part of the messaging toolkit.

Appendix B, "The CoEd Demonstration Program," gives the ToolTalk-related portions of the ptype, header, and `.c` files of the ToolTalk demo program CoEd.

Appendix C, "New ToolTalk Functions," describes the ToolTalk functions that map filenames between local and canonical paths.

Appendix D, "Examples," illustrates the functionality defined in the Tooltalk Messaging Overview book with code examples.

Related Books and Other Documentation

TED ToolTalk Messaging Overview does *not* provide in-depth information about ToolTalk and its functionality. In addition to the ToolTalk product base documentation (that is, *Programmer's Guide*), the following related ToolTalk documentation provide in-depth information about the ToolTalk functionality that is beyond the scope of this book:

- *The ToolTalk Service - An Inter-Operability Solution*
  (Published by SunSoft Press/PTR Prentice Hall, ISBN 013-088717-X)
  This book describes ToolTalk and its functionality in depth, and is appropriate for all platforms to which ToolTalk has been ported. It is available at your local bookstore or directly from PTR Prentice Hall.

- *ToolTalk and Open Protocols*
  by Astrid M. Julienne and Brian Holtz
  (Published by SunSoft Press/PTR Prentice Hall, ISBN 013-031055-7)
  This book describes how to create and develop open protocols for applications that use a messaging service to communicate with other applications. The general principles described in this book provide an application with the flexibility required for users to easily inter-change tools. It is available at your local bookstore or directly from PTR Prentice Hall.

- ToolTalk Message Sets

ToolTalk Desktop Services Message Set	These conventions apply to any tools in a POSIX or X11 environment. In addition to standard messages for these environments, the Desktop conventions define data types and error codes that apply to all of the ToolTalk inter-client conventions.
ToolTalk Document and Media Exchange Message Set	Allows a tool to be a container for arbitrary media, or to be a media player/editor that can be driven from such a container.
CASE Inter-Operability Message Set	An *open* specification defining abstract, framework-neutral message interfaces for CASE set-up by Sunsoft, DEC, and SGI. This

work has been merged with HP's CASE Communique work, which defined message interfaces for HP's SoftBench Broadcast Message Server framework, and was submitted as a joint draft to ANSI X3H6. As of this writing, ANSI X3H6 is still reviewing the joint submission draft.

More information on the draft X3H6 standard can be retrieved from *ftp.netcom.com*, in /pub/X3H6; or you can contact:

X3 Secretariat
Computer and Business Equipment Manufactures Assoc
1250 Eye St NW
Washington DC 20005-3922

Telephone: (202) 737-8888 (press '1' twice)
Fax: (202) 638-4922 or (202) 628-2829

ToolTalk News Group

The ToolTalk news group is:

alt.soft-sys.tooltalk

Introducing the ToolTalk Service 1

As computer users increasingly demand that independently developed applications work together, inter-operability is becoming an important theme for software developers. By cooperatively using each other's facilities, inter-operating applications offer users capabilities that would be difficult to provide in a single application. The ToolTalk service is designed to facilitate the development of inter-operating applications that serve individuals and work groups.

The ToolTalk service enables independent applications to communicate with each other without having direct knowledge of each other. Applications create and send ToolTalk messages to communicate with each other. The ToolTalk service receives these messages, determines the recipients, and then delivers the messages to the appropriate applications, as shown in Figure 1-1.

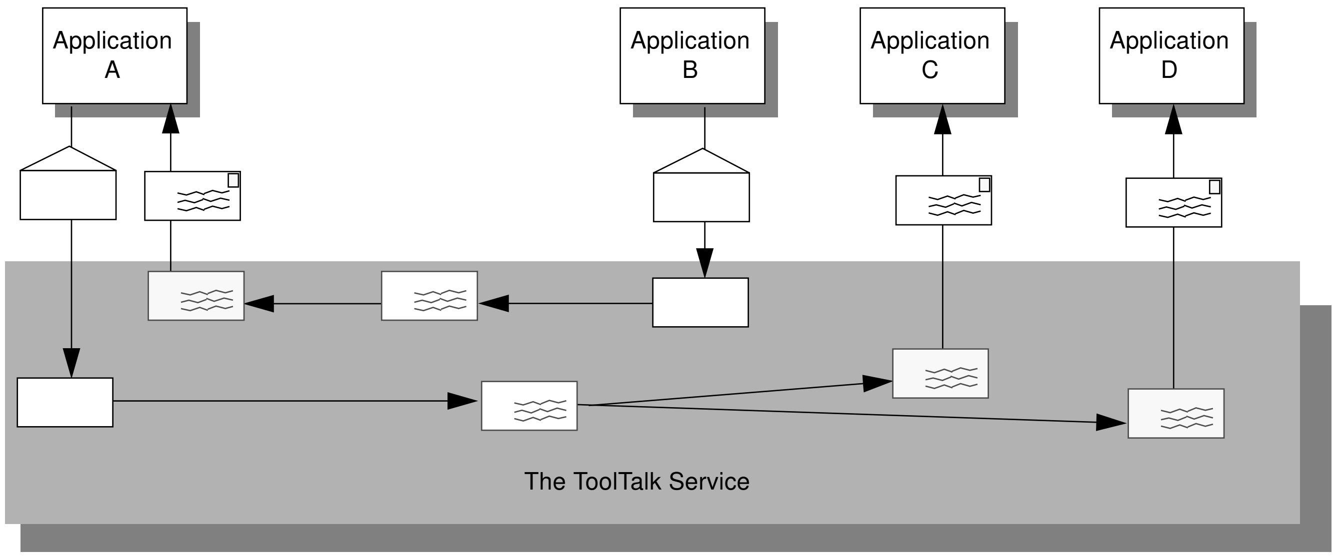

Figure 1-1 Applications Using the ToolTalk Service

What Kind of Work Problems Can the ToolTalk Service Solve?

This section describes some of the inter-operability problems the ToolTalk service is designed to solve. The ToolTalk service is the appropriate technology to use if your application needs:

- Tool inter-changeability.
- Control integration.
- Network-transparent events that are not owned by any well-known server (for example, an X server) and that do not have any predictable set of listeners.
- Automatic tool invocation.
- A widely-available distributed object system.
- Persistent objects.

Of course, there are some inter-operability problems for which the ToolTalk service may not be the appropriate technology to use. However, when your application needs to solve both sorts of problems (that is, a combination of those inter-operability problems for which the ToolTalk service is designed to solve and those problems for which it is not designed), you can use the ToolTalk service in combination with other technologies.

Tool Inter-changeability

Use the ToolTalk service when you want plug-and-play capability. The term *plug-and-play* means that any tool can be replaced by any other tool that follows the same protocol. That is, any tool that follows a given ToolTalk protocol can be placed (plugged) into your computing environment and perform (play) those functions indicated by the protocol. Tools can be mixed and matched, without modification and without having any specific built-in knowledge of each other.

Control Integration

Use the ToolTalk service when your application requires control integration. The term *control integration* indicates a group of tools working together toward a common end without direct user intervention. The ToolTalk service enables control integration through its easy and flexible facilities for issuing arbitrary requests, either to specific tool instances or to anonymous service providers.

Network-Transparent Events

Use the ToolTalk service when your application needs to generate or receive network-transparent events. To be useful, traditional event mechanisms (such as signals and window-system events) require special circumstances; for example, you must know a process or window ID. The ToolTalk service allows events to be expressed naturally: in terms of the file to which the event refers, or the group of processes on the network to which the event is applicable. The ToolTalk service delivers events (called *notices*) to any interested process anywhere on the network. ToolTalk notices are a flexible and easy way to provide extensibility for your system.

Automatic Tool Invocation

Use the ToolTalk service when your application needs network-transparent automatic invocation. The ToolTalk service lets you describe the messages that, when sent from any location on the network, should cause your tool to be invoked. The ToolTalk auto-start facility is easier to use and less host-specific than the conventional `inetd(1)` facility.

Distributed-Object System

Use ToolTalk when you need to build your application on a distributed-object system that is available across a wide variety of platforms. ToolTalk's object system can be used by any application on all the popular UNIX platforms, regardless of whether the application,

- is single- or multi-threaded,
- has a command-line or graphical user interface,
- uses its own event loop, or that of a window-system toolkit.

Note – Programs coded to the ToolTalk object-oriented messaging interface are *not* portable to CORBA-compliant systems without source changes.

Persistent Objects

Use the ToolTalk service when your application needs to place objects unobtrusively in the UNIX file system.

Scenarios Illustrating How the ToolTalk Service Helps Solve Work Problems

The scenarios in this section illustrate how the ToolTalk service helps users solve their work problems. The message protocols used in these scenarios are hypothetical.

Using the ToolTalk Desktop Services Message Set

The ToolTalk Desktop Services Message Set allows an application to integrate and control other applications without user intervention. This section presents two scenarios (“The Smart Desktop” and “Integrated Toolsets” on page 6) that show how the Desktop Services Message Set might be implemented.

The Smart Desktop

Note – The scenario in this section is intended to illustrate how the ToolTalk service can be used in an application-level program that interprets user requests; it is *not* intended to illustrate how the Common Desktop Environment product implements the ToolTalk service to interpret user requests.

A common user requirement for a graphic user interface (GUI) front-end is the ability to have data files be aware (or “know”) of their applications. To do this, an application-level program is needed to interpret the user’s requests. Examples of application-level programs (known as *smart desktops*) are the Apple Macintosh® finder, Microsoft Windows™ File Manager, and the TriTeal Enterprise Desktop File Manager. The key common requirements for smart desktops are:

1. Takes a file.
2. Determines its application.
3. Invokes the application.

The ToolTalk Service provides additional flexibility by allowing classes of tools to edit a specific data type. The following scenario illustrates how the Desktop Services Message Set might be implemented as a smart desktop transparent to the end-user.

1. Diane double-clicks on the File Manager icon.

- The File Manager opens and displays the files in Diane's current directory.

2. Diane double-clicks on an icon for a data file.

   a. The File Manager requests that the file represented by the icon be displayed. The File Manager encodes the file type in the *display* message.

   b. The ToolTalk session manager matches the pattern in the *display* message to a registered application (in this case, the Icon Editor), and finds an instance of the application running on Diane's desktop.

Note – If the ToolTalk session manager does not find a running instance of the application, it checks the statically-defined process types (ptypes) and starts an application that best matches the pattern in the message. If none of the ptypes matches, the session manager returns failure to the File Manager application.

---

   c. The Icon Editor accepts the *display* message, de-iconifies itself, and raises itself to the top of the display.

3. Diane manually edits the file.

Integrated Toolsets

Another significant application for which the Desktop Services Message Set can be implemented is *integrated toolsets*. These environments can be applied in vertical applications (such as a CASE developer toolset) or in horizontal environments (such as compound documents). Common to both of these applications is the premise that the overall solution is built from specialized applications designed to perform one particular task well. Examples of integrated toolset applications are text editors, drawing packages, video or audio display tools, compiler front-ends, and debuggers. The integrated toolset environment requires applications to interact by calling on each other to handle user requests. For example, to display video, an editor calls a video display program; or to check a block of completed code, an editor calls a compiler.

The following scenario shows how the Desktop Services Message Set might be implemented as an integrated toolset:

1. Bruce is working on a compound document using his favorite editor. He decides to change the some of the source code text.

2. Bruce double-clicks on the source code text.
    a. The Document Editor first determines the text represents source code and then determines which file contains the source code.
    b. The Document Editor sends an *edit* message request, using the file name as a parameter for the message.
    c. The ToolTalk session manager matches the pattern in the *edit* message to a registered application (in this case, the Source Code Editor), and finds an instance of the application running on Bruce's desktop.

Note – If the ToolTalk session manager does not find a running instance of the application, it checks the statically-defined ptypes and starts an application that best matches the pattern in the message. If none of the ptypes matches, the session manager returns failure to the Document Editor application.

---

    d. The Source Code Editor accepts the *edit* message request.
    e. The Source Code Editor determines that the source code file is under configuration control, and sends a message to check out the file.
    f. The Source Code Control application accepts the message and creates a read-write copy of the requested file. It then passes the name of the file back to the Source Code Editor.
    g. The Source Code Editor opens a window that contains the source file.
3. Bruce edits the source code text.

Using the ToolTalk Document and Media Exchange Message Set

The ToolTalk Document and Media Exchange Message Set is very flexible and robust. This section illustrates three uses of the ToolTalk Document and Media Exchange Message Set:

- Integrating multimedia into an authoring application.
- Adding multimedia extensions to an existing application.
- Extending the *cut-and-paste* facility of X with a media-translation facility.

Integrating Multimedia Functionality

Integrating multimedia functionality into an application allows end-users of the application to embed various media types in their documents.

Typically, an icon that represents the media object is embedded in the document. Upon selection of an embedded object, the ToolTalk service automatically invokes an appropriate external media application and the object is played as illustrated in the following scenario.

1. Daniel opens a document that contains multimedia objects.
2. The window shows the document with several icons representing various media types (such as sound, video, and graphics).
3. Daniel double-clicks on the sound icon.
   A sound application (called a *player*) is launched and the embedded recording is played.
4. To edit the recording, Daniel clicks once on the icon to select it and uses the third mouse button to display an Edit menu.
   An editing application is launched, and Daniel edits the media object.

Adding Multimedia Extensions to Existing Applications

The ToolTalk Document and Media Exchange Message Set also allows an application to use other multimedia applications to extend its features or capabilities. For example, a Calendar Manager can be extended to use the Audio Tool to play a sound file as a reminder of an appointment, as illustrated in the following scenario:

1. Shelby opens her Calendar Manager and sets an appointment.
2. Shelby clicks on an Audio Response button, which causes the Audio Tool to start.
3. Shelby records her message; for example, “Bring the report.”

When Shelby’s appointment reminder is executed, the Calendar Manager will start the Audio Tool and play Shelby’s recorded reminder.

Extending the X Cut-and-Paste Facility

The ToolTalk Document and Media Exchange Message Set can support an extensible, open-ended translation facility. The following scenario illustrates how an extensible multimedia *cut and paste* facility could work:

1. Maria opens two documents that are different media types.
2. Maria selects a portion of Document A and cuts the portion using the standard *X*-windowing *cut* facility.
3. Maria then pastes the cut portion into Document B.
   a. Document B negotiates the transfer of the cut data with Document A.
   b. If Document B does not understand any of the types offered by Document A, it requests that Document A sends it a *tagged media type*. Document B uses the tagged media type to broadcast a ToolTalk message requesting a translation of the media type to a media type it understands.
   c. A registered translation utility accepts the request and returns the translated version of the media type to Document B.
   d. The paste of the translated data into Document B is performed.

How Applications Use ToolTalk Messages

Applications create, send, and receive ToolTalk messages to communicate with other applications. Senders create, fill in, and send a message; the ToolTalk service determines the recipients and delivers the message to the recipients. Recipients retrieve messages, examine the information in the message, and then either discard the message or perform an operation and reply with the results.

Sending ToolTalk Messages

ToolTalk messages are simple structures that contain fields for address, subject, and delivery information. To send a ToolTalk message, an application obtains an empty message, fills in the message attributes, and sends the message. The sending application needs to provide the following information:

- Is the message a notice or a request (that is, should the recipient respond to the message)?

- What interest does the recipient share with the sender? (For example, is the recipient running in a specific user session or interested in a specific file?)

To narrow the focus of the message delivery, the sending application can provide more information in the message.

Message Patterns

An important ToolTalk feature is that senders need to know little about the recipients because applications that want to receive messages explicitly state what message they want to receive. This information is registered with the ToolTalk service in the form of *message patterns*.

Applications can provide message patterns to the ToolTalk service at installation time and while the application is running. Message patterns are created similarly to the way a message is created; both use the same type of information. For each type of message an application wants to receive, it obtains an empty message pattern, fills in the attributes, and registers the pattern with the ToolTalk service. These message patterns usually match the message protocols that applications have agreed to use. Applications can add more patterns for individual use.

When the ToolTalk service receives a message from a sending application, it compares the information in the message to the register patterns. Once matches have been found, the ToolTalk service delivers copies of the message to all recipients.

For each pattern that describes a message an application wants to receive, the application declares whether it can *handle* or *observe* the message. Although many applications can observe a message, only one application can handle the message to ensure that a requested operation is performed only once. If the ToolTalk service cannot find a handler for a request, it returns the message to the sending application indicating that delivery failed.

Receiving ToolTalk Messages

When the ToolTalk service determines that a message needs to be delivered to a specific process, it creates a copy of the message and notifies the process that a message is waiting. If a receiving application is not running, the ToolTalk service looks for instructions (provided by the application at installation time) on how to start the application.

The process retrieves the message and examines its contents.

- If the message contains a notice that an operation has been performed, the process reads the information and then discards the message.
- If the message contains a request to perform an operation, the process performs the operation and returns the result of the operation in a reply to the original message. Once the reply has been sent, the process discards the original message.

ToolTalk Message Distribution

The ToolTalk service provides two methods of addressing messages: *process-oriented* messages and *object-oriented* messages.

Process-Oriented Messages

Process-oriented messages are addressed to processes. Applications that create a process-oriented message address the message to either a specific process or to a particular type of process. Process-oriented messages are a good way for existing applications to begin communication with other applications. Modifications to support process-oriented messages are straightforward and usually take a short time to implement.

Object-Oriented Messages

Object-oriented messages are addressed to objects managed by applications. Applications that create an object-oriented message address the message to either a specific object or to a particular type of object. Object-oriented messages are particularly useful for applications that currently use objects or that are to be designed around objects. If an existing application is not object-oriented, the ToolTalk service allows applications to identify portions of application data as objects so that applications can begin to communicate about these objects.

Note – Programs coded to the ToolTalk object-oriented messaging interface are *not* portable to CORBA-compliant systems without source changes.

Determining Message Delivery

To determine which groups receive messages, you *scope* your messages. Scoping limits the delivery of messages to a particular session or file.

Sessions

A *session* is a group of processes that have an instance of the ToolTalk message server in common. When a process opens communication with the ToolTalk service, a default session is located (or created, if a session does not already exist) and a *process identifier* (*procid*) is assigned to the process. Default sessions are located either through an environment variable (called “process tree sessions”) or through the X display (called “X sessions”).

The concept of a session is important in the delivery of messages. Senders can scope a message to a session and the ToolTalk service will deliver it to all processes that have message patterns that reference the current session. To update message patterns with the current *session identifier* (sessid), applications join the session.

Files

A container for data that is of interest to applications is called a *file* in this book.

The concept of a file is important in the delivery of messages. Senders can scope a message to a file and the ToolTalk service will deliver it to all processes that have message patterns that reference the file without regard to the process’s default session. To update message patterns with the current file path name, applications join the file.

You can also scope a message to a file within a session. The ToolTalk service will deliver the message to all processes that reference both the file and session in their message patterns.

Note – The file scoping feature is restricted to NFS® and UFS file systems.

Modifying Applications to Use the ToolTalk Service

Before you modify your application to use the ToolTalk service, you must define (or locate) a ToolTalk *message protocol*: a set of ToolTalk messages that describe operations applications agree to perform. The message protocol specification includes the set of messages and how applications should behave when they receive the messages.

To use the ToolTalk service, an application calls ToolTalk functions from the ToolTalk API. The ToolTalk API provides functions to register with the ToolTalk service, to create message patterns, to send messages, to receive messages, to examine message information, and so on. To modify your application to use the ToolTalk service, you must first include the ToolTalk API header file in your program. You also need to modify your application to:

- Initialize the ToolTalk service and join a session.
- Register message patterns with the ToolTalk service.
- Send and receive messages.
- Unregister message patterns and leave your ToolTalk session.

How to Use ToolTalk Messaging 2

Note – The code examples shown in this chapter are taken from a ToolTalk demo program called CoEd. See Appendix B, "The CoEd Demonstration Program," for a listing of the source code showing how ToolTalk-related code is included in the header and `.c` files for this program.

Telling Your Application about ToolTalk Functionality

Before your application can utilize the inter-operability functionality provided by the ToolTalk service and the Messaging Toolkit, it needs to know where the ToolTalk libraries and toolkit reside.

Using the Messaging Toolkit and Including ToolTalk Commands

To use the ToolTalk service, an application calls ToolTalk functions from the ToolTalk API. The Messaging Toolkit provides functions such as functions to register with the ToolTalk service, to create message patterns, to send messages, to receive messages, and to examine message information. To modify your application to use the ToolTalk service and toolkit, you must include the appropriate header files in your application's `.h` file.

```
#include <Tt/tt_c.h>          // ToolTalk Header File
#include <Tt/tttk.h>          // Messaging Toolkit Header File
```

Your application also needs to know about the new ToolTalk commands that are in its `.c` file. Place this information in your application's `.h` file, too.

The code example below shows how the header file information is included in the `CoEditor.h` file.

```
#ifndef CoEditor_h
#define CoEditor_h

#include <X11/Intrinsic.h>
#include <Tt/tt_c.h>                    // ToolTalk Header
#include <Tt/tttk.h>                    // Messaging Toolkit Header
```

Using the ToolTalk Libraries

You need to change the makefile of your application so that it uses the ToolTalk libraries. To do this, add the `-ltt` option as follows:

```
LOCAL_LIBRARIES = -ltt $(XAWLIB) $(XMULIB) $(XTOOLLIB) $(XLIB)
```

Before You Start Coding

Before you can incorporate the Messaging Toolkit functionality into your application, you need to determine the way that your tool will work with other tools. There are several basic questions you need to ask:

1. How will these tools work together?
2. What kinds of operations can these tools perform?
3. What kinds of operations can these tools ask other tools to perform?
4. What events will these tools generate which may interest other tools? (What types of messages will these tools want to send?)
5. What events generated by other tools will be of interest to these tools? (What types of messages will these tools want to receive?)

To best answer these questions, you need to understand the difference between events and operations, and how the ToolTalk service handles messages regarding each of these.

What Is the Difference Between an Event and an Operation?

An *event* is an announcement that something has happened. An event is simply a news bulletin. The sending process has no formal expectations as to whether any other process will hear about the event, or whether an action is taken as a consequence of the event. When a process uses the ToolTalk service to inform interested processes that an event has occurred, it sends a *notice*. Since the sending process does not expect a reply, an event cannot fail.

An *operation* is an inquiry or an action. The requesting process makes an inquiry or requests that an operation be performed. The requesting process expects a result to be returned and needs to be informed of the status of the inquiry or action. When a process uses the ToolTalk service to ask another tool to perform an operation, it sends a *request*. The ToolTalk service delivers the request to interested processes and informs the sending process of the status of the request.

Sending Notices

When your application sends a ToolTalk notice, it will not receive a reply or be informed about whether or not any tool pays attention to the notice. It is important to make the notice an impartial report of the event as it happens.

For example, if your tool sends the Desktop Services message `Modified`, it may expect any listening tools to react in a given way. However, your tool should not care, and does not need to be informed, about whether any or no other tool reacts to the message; it only wants to report the event:

```
THE_USER_HAS_MADE_CHANGES_TO_THIS.
```

Sending Requests

When your application sends a ToolTalk request, it expects one tool to perform the indicated operation, or to answer the inquiry, and return a reply message. For example, if your tool sends the Desktop Services message `Get_Modified`, it should expect notification that the message was delivered and the action performed. The ToolTalk service guarantees that either a reply will be returned by the receiving process or the sender will be informed of the request’s failure.

You can identify requests in three ways:

1. By identifying the operations requested by your tool that can fail.

2. By identifying the operations your tool can perform for other tools.
3. By identifying the operations your tool will want other tools to perform.

A good method to use to identify these operations is to develop a scenario that outlines the order of events and operations that you expect your tool to perform and have performed.

Developing a Scenario

A scenario outlines the order of the events and operations that a tool will expect to perform and have performed. For example, the following scenario outlines the events that the ToolTalk demo program CoEd expects to perform and have performed:

1. User double-clicks on a document icon in the File Manager.
   The file opens in the editor, which is started by the system if one is not already running.

   If another tool has modifications to the text pending for the document, User is asked whether the other tool should save the text changes or revert to the last saved version of the document.
2. User inserts text.
3. User saves the document.
   If another tool has modifications pending for the document, User is asked whether to modify the document.
4. User exits the editor.
   If text has unsaved changes, User is asked whether to save or discard the changes before quitting the file.

Once the scenario is done, you can answer your basic questions.

How Will the Tools Work Together?

- The File Manager will request that CoEd open a document for editing.
- Each instance of CoEd will notify other interested instances of changes it makes to the state of the document.

What Kinds of Operations Do the Tools Perform?

- Each instance of CoEd can answer questions about itself and its state, such as “What is your status?”
- Each instance of CoEd has the capability of performing operations such as:
  - Iconifying and de-iconifying.
  - Raising to front and lowering to back.
  - Editing a document.
  - Displaying a document.
  - Quitting.

What Kinds of Operations Can the Tools Ask Other Tools to Perform?

- The File Manager must request that CoEd open a document for editing.
- An instance of CoEd can ask another instance of CoEd to save changes to the open document.
- An instance of CoEd can ask another instance of CoEd to revert to the last saved version of the open document.

What Events Will the Tools Generate that May Interest Other Tools?

- The document has been opened.
- The document has been modified.
- The document has been reverted to last saved version.
- The document has been saved.
- An instance of CoEd has been exited.

What Events Generated by Other Tools Will Be of Interest to This Tool?

- The document has been opened.
- The document has been modified.
- The document has been reverted to last saved version.
- The document has been saved.
- An instance of CoEd has been exited.

Preparing Your Application for Communication

The ToolTalk service provides you with a complete set of functions for application integration. Using the functionality provided with the ToolTalk Messaging Toolkit, your applications can be made to “speak” to other applications that are ToolTalk-compliant. This section describes how to add the kinds of ToolTalk functions you need to include in your application so that it can communicate with other ToolTalk-aware applications that follow the same protocols.

Creating a Ptype File

The ToolTalk types mechanism is designed to help the ToolTalk service route messages. When your tool declares a ptype, the message patterns listed in it are automatically registered; the ToolTalk service then matches messages it receives to these registered patterns. These static message patterns remain in effect until the tool closes communication with the ToolTalk service.

The ToolTalk Types Database already has installed ptypes for tools bundled with this release. You can extract a list of the installed ptypes from the ToolTalk Types Database, as follows:

`% tt_type_comp -d` *user*|*system*|*network* `-P`

The names of the ptypes will be printed out in source format.

For all other tools (that is, tools that are not included in this release), you need to first create a ptype file to define the ptype for your application, and then compile the ptype with the ToolTalk type compiler, `tt_type_comp`. To define a ptype, you need to include the following information in a file:

- A process-type identifier (*ptid*).
- An optional start string – The ToolTalk service will execute this command, if necessary, to start a process running the program.
- Signatures – Describes the `TT_PROCEDURE`-addressed messages that the program wants to receive. Messages to be observed are described separately from messages to be handled.

To create a ptype file, you can use any text editor (such as `vi`, `emacs`, or `dtpad`). The code example below shows a snippet from the ptype file for the CoEd application.

```
ptype DT_CoEd {                        /* Process type identifier */
    start "CoEd";                      /* Start string */
    handle:                            /* Receiving process */
    /*
     * Display ISO_Latin_1
     */
```

```
    session        Display( in    ISO_Latin_1 contents) => start
opnum = 1;                             /* Signature */
        /* NOTE: A signature is divided
         * into two parts by the => as follows:
         * Part 1 specifies how the message is to be matched;
         * Part 2 specifies what is to be taken when
         * a match occurs.
         */
}
```

After you have created the ptype file, you need to install the ptype. To do this, run the ToolTalk type compiler. On the command line, type the following:

`% tt_type_comp` *CoEd.ptype*

where *CoEd.ptype* is the name of the CoED ptype file.

Testing for Existing Ptypes in Current Session

The ToolTalk service provides a simple function to test if a given ptype is already registered in the current session.

```
// Test for existing ptype registered in current session
tt_ptype_exists(const char *ptid)
```

where *ptid* is the identifier of the session to test for registration.

Merging a Compiled Ptype File into a Currently Running ttsession

The ToolTalk service provides a function to merge a compiled ToolTalk type file into the currently running `ttsession`:

```
// Merge new compiled ptypes into currently running ttsession
tt_session_types_load(current_session, compiled_types_file)
```

where *current_session* is the current default ToolTalk session and *compiled_types_file* is the name of the compiled ToolTalk types file. This function adds new types and replaces existing types of the same name; other existing types remain unchanged.

Tasks Every ToolTalk-aware Application Needs to Perform

There are a number of tasks every ToolTalk-aware application needs to perform, including:

- Initializing the toolkit.
- Joining a ToolTalk session and registering patterns.
- Adding the ToolTalk service to its event loop.

This section provides examples of the ToolTalk code you need to include in your application so that it can perform these tasks.

Note – The code snippets used in this section are taken from the `CoEd.C` file. This file contains the general commands any application needs to perform that are not specific to any particular application. See Appendix B, "The CoEd Demonstration Program," for the detailed source code.

Initializing the Toolkit

Your application needs to initialize and register with the initial ToolTalk session. To do so, it first needs to obtain a process identifier (procid). The following code snippet shows how to obtain a procid and how to initialize the toolkit.

```
// Obtain process identifier
    int myTtFd;
// Initialize toolkit and create a ToolTalk communication endpoint
    char *myProcID = ttdt_open( &myTtFd, ToolName, "SunSoft",
                                "%I", 1 );
```

Caution – Your application *must* call `ttdt_open` before any other calls are made; otherwise, errors may occur.

Joining the ToolTalk Session and Registering Message Patterns

Before your application can receive messages, it must join a ToolTalk session and register the message patterns that are to be matched.

```
// Join a ToolTalk session and register patterns
```

```
// and default callbacks
        sessPats = ttdt_session_join( 0, 0, session_shell,
                                      this, 1 );
```

Adding the ToolTalk Service to Event Loop

Your application also needs to add the ToolTalk service to its event loop.

```
// Process ToolTalk events for Xt Clients
       XtAppAddInput( myContext, myTtFd, (XtPointer)XtInputReadMask,
                      tttk_Xt_input_handler, myProcID );
```

Tasks ToolTalk-aware Editor Applications Need to Perform

In addition to the duties described in the section "Tasks Every ToolTalk-aware Application Needs to Perform," ToolTalk-aware editor applications also need to perform other tasks, including:

- Declaring a ptype.
- Processing the start string message.
- Passing a media callback.
- Failing a message.
- Replying when a request has been completed.

This section provides examples of the ToolTalk code you need to include in your editor application so that it can perform these additional tasks.

Note – The code snippets used in this section are taken from the `CoEditor.C` file. This file contains specific commands for editor applications. See Appendix B, "The CoEd Demonstration Program," for the detailed source code.

Writing a Media Load Pattern Callback

There is one step you need to perform before you code your editor application to include any ToolTalk functions: you need to write a media load pattern callback routine. For example,

```
Tt_message
```

```
CoEditor::loadISOLatin1_(
    Tt_message        msg,
    void              *pWidget,
    Ttttk_op          op,
    Tt_status         diagnosis,
    unsigned char     *contents,
    int               len,
    char              *file,
    char              *docname
)
```

This callback is passed to the media load function at runtime.

Declaring a Ptype

Since type information is specified only once (when your application is installed), your application needs to only declare its ptype each time it starts.

Passing Media Load Pattern Callbacks

The media load pattern callback routine you wrote previously is passed in at runtime. The callbacks are registered when your application joins the session. When your tool agrees to handle a request, a callback message is sent. A callback message is also sent if a file is joined or if a message is failed.

```
// Join the session and register patterns and callbacks
sessPats = ttdt_session_join( 0, 0, session_shell, this, 1 );

// Accept responsibility to handle a request
    _contractPats = ttdt_message_accept(msg,
                              CoEditor::_contractCB_,
                              shell, this, 1, 1 );

// Optional task: Join a file (Can be called recursively)
    if (_filePats == 0) {_filePats = ttdt_file_join( _file,
                              TT_SCOPE_NONE, 1,
                              CoEditor::_fileCB_, this );
    }

// Fail a message
tttk_message_fail( msg, TT_DESKTOP_ENODATA, 0, 1 );
```

Replying When Request Is Completed

After your application has completed the operation request, it must reply to the sending application. The following message returns the edited contents of text to the sender.

```
// Reply to media load pattern callback
// with edited contents of text
ttmedia_load_reply( _contract, (unsigned char *)contents,
                    len, 1 );
```

Optional Tasks ToolTalk-aware Editor Applications Can Perform

In addition to the tasks described in the section "Tasks ToolTalk-aware Editor Applications Need to Perform," editor applications can also perform other optional tasks such as tasks that use desktop file interfaces to coordinate with other editors. This section provides examples of some of the ToolTalk code you need to include in your editor application so that it can perform these optional tasks.

Note – The code snippets used in this section are taken from the `CoEditor.C` file. This file contains specific commands for editor applications. See Appendix B, "The CoEd Demonstration Program," for the detailed source code.

Requesting Modify, Revert, or Save Operations

The following code snippet asks a file whether it has any changes pending:

```
// Does the file have any changes pending?
_modifiedByOther = ttdt_Get_Modified( _contract, _file, TT_BOTH,
                       10 * timeOutFactor );
```

The following code snippet reverts a file to its last version:

```
// Revert file to last version
status = ttdt_Revert( _contract, _file, TT_BOTH,
              10 * timeOutFactor );
```

The following code snippet saves pending changes to a file:

```
// Save pending changes
status = ttdt_Save( _contract, _file, TT_BOTH,
            10 * timeOutFactor );
```

Notifying When a File Is Modified, Reverted, or Saved

The following code snippet announces to interested tools that your application has changes pending for the file:

```
// File has been modified
ttdt_file_event( _contract, TTDT_MODIFIED, _filePats, 1 );
```

The following code snippet announces to interested tools that your application has reverted the file to its last saved version:

```
// File has been reverted to last version
ttdt_file_event( _contract, TTDT_REVERTED, _filePats, 1 );
```

The following code snippet announces to interested tools that your application has saved its pending changes for the file.

```
// File has been saved
ttdt_file_event( _contract, TTDT_SAVED, _filePats, 1 );
```

Quitting a File

The following code snippet unregisters interest in ToolTalk events about a file and destroys the patterns.

```
// Unregister interest in ToolTalk events and destroy patterns
status = ttdt_file_quit( _filePats, 1 );
    _filePats = 0;
```

Using TTSnoop to Debug Messages and Patterns

3

TTSnoop is a tool provided to create and send custom-constructed ToolTalk messages. You can also use TTSnoop as a tool to selectively monitor any or all ToolTalk messages.

About TTSnoop

TTSnoop is a useful interactive tool that you can use to become familiar with TookTalk concepts and API calls as well as to perform demonstrations. In addition, TTSnoop is a valuable debugging tool when you are developing applications.

You can use TTSnoop to monitor for messages that match more than one pattern. When a matched message is displayed, the name of the pattern that matched the entry can also be displayed.

You can add, edit, or delete messages and patterns to scrollable lists. TTSnoop allows the definitions of multiple patterns and messages to be saved and loaded from files. You can also define, save, and reload patterns and messages particular to a category of applications (for example, DeskSet™ tools) as well as associate messages and patterns with a user-defined name.

Where to Find TTSnoop

The TTSnoop program resides in the directory `/usr/dt/bin/ttsnoop`.

Starting TTSnoop

To start the program, enter the following command on the command line:

`ttsnoop` [-t]

The `-t` option displays the ToolTalk API calls that are being used to construct a particular pattern or message. Table 3-1 describes the buttons that are displayed when TTSnoop starts.

Table 3-1 TTSnoop Buttons

Button	Description
Start	Click this button to activate message reception. TTSnoop will display any incoming messages that match the patterns you register.
Stop	Click this button to stop receiving messages.
Clear	Click this button to clear the window.
About TTSnoop	Click this button to obtain general help for TTSnoop.
Display	Click this button to display a panel of checkboxes to highlight specific ToolTalk message components on the TTSnoop display subwindow.
Messages	Click this button to display a panel that enables you to create, store, and send ToolTalk messages.
Patterns	Click this button to display a panel that enables you to compose and register ToolTalk patterns.
Send Messages	Click this button to send messages that were stored using the Messages display.

Note – To obtain help for individual buttons and settings, place the mouse over the button or setting and click the F1 key or Help key on your keyboard.

Composing and Sending Messages

When you click the Messages button on the main display window, a display panel containing the choices shown in Table 3-2 is displayed.

Table 3-2 Message Button Display Window Options

Button	Description
Add Message	Click this button to store the current message settings. Once the messages are stored, you can recall and send these messages using the Send Message button on the main display window.
Edit Contexts	Click this button to add, change, and delete send message contexts. The display window displayed allows you to edit contexts to be sent with your messages.
Send Message	Click this button to send the newly created message.

Composing and Registering Patterns

When you click the Patterns button on the main display window, a display panel is displayed.

Click the Apply button to register your pattern. Once a pattern is registered, you can use TTSnoop as a debugging tool to observe what messages are being sent by other applications.

Click the Edit Receive Contexts button to add, change, and delete receive message contexts in patterns. The window displayed enables you to edit contexts to be registered with your patterns.

Displaying Message Components

When you click the Display button on the main display window, a display panel of checkboxes is displayed.

When you select a checkbox and click the Apply button, the specified ToolTalk message component is displayed until you make another selection and apply the change.

Sending Pre-Created Messages

When you click the Send Message button on the main display window, you can send one of the messages you created and stored using the Messages display.

Receiving Messages

When you click the Start button on the main display window, TTSnoop will display any incoming messages that match the patterns you registered.

Stop Receiving Messages

When you click the Stop button on the main display window, TTSnoop will stop receiving messages.

Using ToolTalk Tracing 4

The ToolTalk ttsession trace shows how ToolTalk pattern matches and delivers every message ttsession sees. ToolTalk tracing for this release also:

- Displays a single client's interactions with ToolTalk. This feature allows implementors to focus on only one client.
- Filters the ttsession trace by, for example, message type, sender, or receiver.

Accessing ToolTalk Tracing

A command new for this release, `tttrace`, is the primary way to access ToolTalk tracing. This command is similar in purpose and command-line interface to the `truss` command. It enables you to control the three kinds of ToolTalk tracing. The `tttrace` command has two fundamental modes: *server* mode and *client* mode.

- In server mode, `tttrace` directs the indicated session to trace by sending it a Session_Trace request.
- In client mode, `tttrace` sets an environment variable and executes the ToolTalk client command given on the command line. The environment variable in the executed client instructs `libtt` whether, and how, to trace client messaging and client API calls.

Note – `tttrace` is not downward compatible with older servers or with clients using older versions of `libtt`. While `tttrace` will detect and diagnose older servers, it fails silently on clients using older versions of `libtt`.

Controlling Tracing

Controlling libtt Tracing

One way to control `libtt`'s tracing behavior is to set the environment variable `$TT_TRACE_SCRIPT`.

Note – `libtt`'s tracing fails if the variable's value is corrupt or inconsistent.

Controlling Client-Side Tracing

The `tt_trace_control` call sets or clears an internal flag to control all client-side tracing. You can use this call to trace suspect areas in your code. The format of this call is:

```
int tt_trace_control(int option)
```

where *option* 0 to turn traciing off; 1 to turn tracing on; and -1 to toggle tracing on and off. When tracing is on, the extent of tracing is controlled by the `TT_TRACE_SCRIPT` variable or tracefile. This call returns the previous setting of the trace flag.

Tracing Message Traffic in a ToolTalk Session

The Session_Trace request is a ToolTalk request that ttsession registers to handle itself; that is, ttsession is the handler for the Session_Trace request. This request can be sent by any ToolTalk client, and, although not recommended, other ToolTalk clients can register to handle this request. (Note: This method will cause tracing to *not* work.) The syntax for this request is:

```
[file]Session_Trace(inbooleanon,
        in  booleanfollow
        [in attributetoPrint
        |in statetoTrace
        |in op  toTrace
        |in handler_ptypetoTrace
        |in sender_ptypetoTrace][...] );
```

The Session_Trace request turns message tracing in the scoped-to session on or off.

- If tracing is on and the file attribute of the request is set, subsequent trace output is appended to the file named by the attribute.
- If tracing is on and the file attribute is *not* set, tracing continues to the current trace.

By default, *daemon* mode causes the output to go to the console of the host on which ttsession is running; job-control mode causes the output to go to ttsession's standard error. Table 4-1 describes the required and optional arguments for this request.

Table 4-1 Session_Trace Arguments

Argument		Description
`boolean` *on*	Required	Turn tracing on or off. If no *toTrace* arguments are included and *on* is true, the previous trace settings are restored.
`boolean` *follow*	Required	Turn on client-side tracing for Invoked clients.
`attribute` *toPrint*	Optional	Print attribute(s) for each message traced. Valid attributes are: • *none*–print only a one-line description of traced messages (default). • *all*–print all attributes of traced messages.
`state` *toTrace*	Optional	State(s) through which to trace messages. In addition to the Tt_states defined in `tt_c.h`, valid *state*s are: • *edge*–messages entering initial (`TT_SENT`) and final (`TT_HANDLED, TT_FAILED`) states. • *deliver*–all state changes and all client deliveries. • *dispatch*–deliver + all patterns considered for matching. (default)

Table 4-1 Session_Trace Arguments *(Continued)*

Argument		Description
`op` *toTrace* `sender_ptype` *toTrace* `handler_ptype` *toTrace*	Optional Optional Optional	Trace messages that have *toTrace* as a value for the indicated message attribute. • Any number of *toTrace* arguments may be included in the request. • *toTrace* may include `sh` wildcard characters. • If no *toTrace* argument is included for a given message attribute, no value of that attribute excludes a message from tracing.

The current session tracing behavior changes only if this request is not failed. On failure, the `tt_message_status` of the reply is set to one of the errors described in Table 4-2.

Table 4-2 Error Messages Returned by Session_Trace Request

Error	Description
`TT_ERR_NO_MATCH`	No handler could be found for the request.
`TT_ERR_APPFIRST + EACCES`	ttsession does not have permission to open or create the trace file.
`TT_ERR_APPFIRST + EISDIR`	The trace file is a directory.
`TT_ERR_APPFIRST + ENOSPC`	There is not enough space in the target file system to create the trace file.
`TT_ERR_APPFIRST + EEXIST`	Tracing is already occurring on another file. ttsession resets the file attribute of the reply to name the existing trace file. To trace to a different file, first turn off tracing to the current trace file.

Tracing ToolTalk Calls and Messages through the Server

The `tttrace` function traces message traffic through the server for the indicated ToolTalk session, or runs a command with ToolTalk client tracing turned on. If neither the session nor the command is given, the default session is traced. By default, tracing terminates when `tttrace` exits. The syntax for this function is:

```
tttrace [-0FCa] [-o outfile ] [-S session | command]
tttrace [-e script | -f scriptfile ] [-S session | command]
```

Table 4-3 describes the `tttrace` options.

Table 4-3 `tttrace` Options

Option	Description
`-0`	Turns off message tracing in session, or runs the specified command without message tracing (that is, with only call tracing).
`-F`	Follows all children forked by the indicated command, or subsequently started in session by ttsession. Normally, only the indicated command or a ttsession instance is traced. When the `-F` option is specified, the process ID is included with each line of trace output to indicate which process generated it.
`-C`	Do not trace client calls into the ToolTalk API. The default is to trace the calls.
`-a`	Prints all attributes, arguments, and context slots of traced messages. The default is to use only a single line when printing a message on the trace output.
`-o` *outfile*	The file to be used for the trace output. For session tracing, output goes to standard output of `tttrace`.
`-S` *session*	The session to trace. Defaults to the default session; that is, the session that `tt_open` would contact.
command	The ToolTalk client command to invoke and trace.
`-e` *script*	The script to be used as a `tttrace` setting.
`-f` *scriptfile*	The file from which to read the `tttrace` settings.

`tttrace` is implemented purely as a ToolTalk client, using the message interface to ttsession and the `TT_TRACE_SCRIPT` environment variable. If this variable is set, it tells `libtt` to turn on client-side tracing as specified in the trace script. If the first character of the value is '`.`' or '`/`', the value is taken to be the path name of file containing the trace script to use; otherwise, the value is taken to be an inline trace script.

Formats of Traced Functions

The following is an example of how a traced ToolTalk function looks.

`[`*pid*`]` *function_name*`(`*params*`) = `*return_value* `(`*Tt_status*`)`

Message Summary Format

The `-a` option prints message attributes after a one-line summary of the message, as follows:

Tt_state Tt_paradigm Tt_class `(`*Tt_disposition in Tt_scope*`)` **:** *status* `==` *Tt_status*

State Change Format

State changes are indicated by the following format:

old_state `=>` *new_state*`.`

Message Delivery Format

Deliveries are indicated by the following indicated:

`Tt_message => procid` *recipient_procid*

Table 4-4 dexplains the messages you may receive during a dispatch trace.

Table 4-4 Reasons for Dispatch Trace

Message	**Explanation**
`tt_message_send`	The message to send.
`tt_message_reject`	The message was rejected.
`tt_message_fail`	The message failed.
`tt_message_reply`	The reply to a message.

Table 4-4 Reasons for Dispatch Trace *(Continued)*

Message	Explanation
`tt_session_join`	The session to join.
`tt_file_join`	The file to join.
`tt_message_reply`	A client called the indicated function.
`tt_message_send_on_exit`	ttsession is dispatching on_exit messages for a client that disconnected before calling `tt_close`.
`tt_message_accept`	ttsession is dispatching messages that had been blocked while a ptype was being started. The started client has now called either `tt_message_accept` or `tt_message_reply` to indicate that the ptype should be unblocked.
`TT_ERR_PTYPE_START`	A ptype instance was started to receive the message, but the start command exited before it connected to ttsession.
`TT_ERR_PROCID`	ttsession lost its connection to the client that was working on this request.
`ttsession -> ttsession`	Another session wants this session to find recipients for the message.
`ttsession <- ttsession`	Another session wants to update (for example, fail) a message originating in this session.

Matching Format

When dispatching is being traced, matching is indicated by one of the following formats:

```
Tt_message & Tt_pattern {
Tt_message & ptype ptid {
Tt_message & otype otid {
```

The pattern or signature is printed, followed by:

```
} == match_score; [/* mismatch_reason */]
```

Examples

This sections contains examples of how to use the `tttrace` function.

Registering a Pattern and Sending a Matching Notice

To register a pattern and send a notice that matches the pattern, type:

`% tttrace -a` *myclientprogram*

The example below shows the results.

```
tt_open() = 0x51708=="7.jOHHM X 129.144.153.55 0" (TT_OK)
tt_fd() = 11 (TT_OK)
tt_pattern_create() = 0x50318 (TT_OK)
tt_pattern_category_set(0x50318, TT_OBSERVE) = 0 (TT_OK)
tt_pattern_scope_add(0x50318, TT_SESSION) = 0 (TT_OK)
tt_pattern_op_add(0x50318, 0x2f308=="Hello World") = 0 (TT_OK)
tt_default_session() = 0x519e0=="X 129.144.153.55 0" (TT_OK)
tt_pattern_session_add(0x50318, 0x519e0=="X 129.144.153.55 0") = 0 (TT_OK)
tt_pattern_register(0x50318) = 0 (TT_OK)
tt_message_create() = 0x51af0 (TT_OK)
tt_message_class_set(0x51af0, TT_NOTICE) = 0 (TT_OK)
tt_message_address_set(0x51af0, TT_PROCEDURE) = 0 (TT_OK)
tt_message_scope_set(0x51af0, TT_SESSION) = 0 (TT_OK)
tt_message_op_set(0x51af0, 0x2f308=="Hello World") = 0 (TT_OK)
tt_message_send(0x51af0) ...
    TT_CREATED => TT_SENT:
        TT_SENT TT_PROCEDURE TT_NOTICE (TT_DISCARD in TT_SESSION): 0 == TT_OK
        id:     0 7.jOHHM X 129.144.153.55 0
        op:     Hello World
        session:X 129.144.153.55 0
        sender: 7.jOHHM X 129.144.153.55 0
= 0 (TT_OK)
tt_message_receive() ...
    Tt_message => procid <7.jOHHM X 129.144.153.55 0>
    TT_SENT TT_PROCEDURE TT_NOTICE (TT_DISCARD in TT_SESSION): 0 == TT_OK
    id:     0 7.jOHHM X 129.144.153.55 0
    op:     Hello World
    session:X 129.144.153.55 0
    sender: 7.jOHHM X 129.144.153.55 0
    pattern:0:7.jOHHM X 129.144.153.55 0
= 0x51af0 (TT_OK)
```

```
To see ttsession’s view of the message flow, type:
% tttrace -a
ttsession’s view
```

```
tt_message_reply:
    TT_SENT => TT_HANDLED:
        TT_HANDLED TT_PROCEDURE TT_REQUEST (TT_DISCARD in TT_SESSION): 0 == TT_OK
        id:     0 2.jOHHM X 129.144.153.55 0
        op:     Session_Trace
        args:   TT_IN string: "> /tmp/traceAAAa002oL; version 1; states"[...]
        session:X 129.144.153.55 0
        sender: 2.jOHHM X 129.144.153.55 0
        pattern:0:X 129.144.153.55 0
        handler:0.jOHHM X 129.144.153.55 0
        Tt_message => procid <2.jOHHM X 129.144.153.55 0>
tt_message_send:
    TT_CREATED TT_PROCEDURE TT_NOTICE (TT_DISCARD in TT_SESSION): 0 == TT_OK
    id:     0 7.jOHHM X 129.144.153.55 0
    op:     Hello World
    session:X 129.144.153.55 0
    sender: 7.jOHHM X 129.144.153.55 0
    TT_CREATED => TT_SENT:
        TT_SENT TT_PROCEDURE TT_NOTICE (TT_DISCARD in TT_SESSION): 0 == TT_OK
        id:     0 7.jOHHM X 129.144.153.55 0
        op:     Hello World
        session:X 129.144.153.55 0
        sender: 7.jOHHM X 129.144.153.55 0
    Tt_message & Tt_pattern {
        id:     0:7.jOHHM X 129.144.153.55 0
        category:TT_OBSERVE
        scopes: TT_SESSION
        sessions:X 129.144.153.55 0
        ops:    Hello World
    } == 3;
    Tt_message => procid <7.jOHHM X 129.144.153.55 0>
```

of *mylientprogram*'s message flow is shown in .

Note – The first message traced will almost always be ttsession's reply to the request sent to it by `tttrace`.

Tracing a Message Flow

To trace the message flow in a specific, non-default session, type:

`% tttrace` **`-S`** *"01 15303 1342177284 1 0 13691 129.144.153.55 2"*

where *"01 15303 1342177284 1 0 13691 129.144.153.55 2"* is the specific, non-default session to be traced.

Settings for ToolTalk Tracing

A `tttrace` script contains settings that control ToolTalk calls and messages. A `tttrace` script consists of commands separated by semicolons or newlines. If conflicting values are given for a setting, the last value is the one used. Table 4-5 describes these commands.

Table 4-5 `tttrace` Script Commands

Command	Description
`version` *n*	The version of the `tttracefile` command syntax used. The current version is 1.
`follow [off | on]`	Sets whether to follow all children forked by the traced client or subsequently started in the traced session. Default is `off`.
`[> | >>]` *outfile*	File to be used for the trace output. By default, trace output goes to standard error. Normal shell interpretation of > and >> applies.
`functions [all | none` | *func...*]	ToolTalk API functions to trace. *func* may include shell wildcard characters. Default is `all`.
`attributes [all | none]`	`none` (default) means use only a single line when printing a message on the trace output; `all` means print all attributes, arguments, and context slots of traced messages.
`states [none | edge | deliver | dispatch |` *Tt_state*`] ...`	State(s) through which to trace messages. In addition to the Tt_states defined in `tt_c.h`, valid *state*s are: • `none` – disable all message tracing. • `edge` – messages entering initial (`TT_SENT`) and final (`TT_HANDLED`, `TT_FAILED`) states. • `deliver` – all state changes and all client deliveries. • `dispatch` – deliver plus all patterns considered for matching (default).

Table 4-5 `tttrace` Script Commands *(Continued)*

Command	Description
`ops` *toTrace*... `sender_ptypes` *toTrace*... `handler_ptypes` *toTrace*...	Trace messages that have *toTrace* as a value for the indicated message attribute. *toTrace* may include shell wildcard characters. If no *toTrace* argument is included for a given message attribute, then no value of that attribute excludes a message from tracing.

The Messaging Toolkit A

The ToolTalk Messaging Toolkit is a higher-level interface of the ToolTalk API. It provides common definitions and conventions to easily integrate basic ToolTalk messages and functionality into an application for optimum inter-operability with other applications that follow the same message protocols.

Although most of the messages in the ToolTalk Messaging Toolkit are the messages in the standard ToolTalk message sets, the functions of the Messaging Toolkit transparently take care of several tasks that would otherwise need to be coded separately. For example, the `ttdt_file_join` function will register a pattern to observe Deleted, Reverted, Moved, and Saved notices for the specified file in the specified scope; it also invokes a callback message.

General Description of the ToolTalk Messaging Toolkit

Inter-operability is an important theme if independently developed applications are to work together. The messages in the toolkit have been agreed upon by developers of inter-operating applications; the protocols form a small, well-defined interface that maximizes application autonomy.

The ToolTalk Messaging Toolkit plays a key role in application inter-operability and offers complete support for messaging. The message protocol specification includes the set of messages and how applications should behave when they receive the messages. These messages can be retrofitted to any existing application to leverage the functionality of the application. You can easily add these messages to existing applications to send, receive, and use shared information.

Tools that follow the ToolTalk messaging conventions will not use the same ToolTalk syntax for different semantics, nor will tools fail to talk to each other because they use different ToolTalk syntax for identical semantics. If these protocols are observed, cooperating applications can be modified, even replaced, without affecting one another.

Most of the messages in the Messaging Toolkit are the messages in the standard ToolTalk message sets. For detailed descriptions of the standard ToolTalk message sets, see the *ToolTalk Programmer's Guide*. Table A-1 lists the functions described in this chapter that partly comprise the ToolTalk Messaging Toolkit.

Table A-1 ToolTalk Messaging Toolkit Functions

Function	Description
`ttdt_close`	Destroys a ToolTalk communication endpoint.
`ttdt_file_event`	Announces an event about a file.
`ttdt_file_join`	Registers to observe ToolTalk events about a file.
`ttdt_file_notice`	Creates and sends a standard ToolTalk notice about a file.
`ttdt_file_quit`	Unregisters interest in ToolTalk events about a file.
`ttdt_file_request`	Creates and sends a standard ToolTalk request about a file.
`ttdt_Get_Modified`	Asks if any ToolTalk client has changes pending on a file.
`ttdt_message_accept`	Accepts the responsibility for handling a ToolTalk request.
`ttdt_open`	Creates a ToolTalk communication endpoint.
`ttdt_Revert`	Requests that a ToolTalk client revert to the last saved version of a file.

Table A-1 ToolTalk Messaging Toolkit Functions *(Continued)*

Function	Description
`ttdt_Save`	Requests that a ToolTalk client save a file.
`ttdt_sender_imprint_on`	Causes a tool to emulate the behavior and characteristics of the specified ToolTalk tool.
`ttdt_session_join`	Joins a ToolTalk session and registers patterns and default callbacks for many standard desktop messages.
`ttdt_session_quit`	Unregisters any patterns and default callbacks registered when session joined, and quits the ToolTalk session.
`ttdt_subcontract_manage`	Manages outstanding requests.
`ttmedia_Deposit`	Sends a Deposit request to checkpoint a document.
`ttmedia_load`	Creates and sends a Media Exchange request to display, edit, or compose a document.
`ttmedia_load_reply`	Replies to a Display, Edit, or Compose request.
`ttmedia_ptype_declare`	Declares the ptype of a Media Exchange media editor.
`tttk_block_while`	Blocks the program while awaiting a condition such as a reply.
`tttk_message_abandon`	Fails or rejects a message, then destroys it.
`tttk_message_create`	Creates a message that conforms to messaging conventions.
`tttk_message_fail`	Fails a message.
`tttk_message_receive`	Retrieves next ToolTalk message.
`tttk_message_reject`	Rejects a message.
`tttk_op_string`	Returns a string for the operation.
`tttk_string_op`	Returns the operation for the string.
`tttk_Xt_input_handler`	Processes ToolTalk events for Xt clients.

Toolkit Conventions

Most of the messaging conventions for the toolkit consist of descriptions of the standard ToolTalk message sets. This section describes conventions not related to any particular standard message set.

Table A-2 Messaging Toolkit Conventions

Field	Description
fileAttrib	Indicates whether the file attribute of the message can or needs to be set. The ToolTalk service allows each message to refer to a file, and has a mechanism (called “file-scoping”) for delivering messages to clients that are “interested in” the named file.
opName	The name of the operation or event (also called “op”). It is important that different tools use the same opName to mean the same thing. Unless a message is a standard one, its opName must be unique; for example, prefix the opName with `Company_Product` (such as *Acme_HoarkTool_Hoark_My_Frammistat*).
requiredArgs	Arguments that must always be included in the message.
optionalArgs	Extra arguments that may be included in a message. Any optional arguments in a message must be in the specified order and must follow the required arguments.
vtype argumentName	A description of a particular argument. A vtype is a programmer-defined string that describes what kind of data a message argument contains. The ToolTalk service uses vtypes only for matching sent message instances with registered message patterns. Every vtype should by convention map to a single, well-known data type.

Using the Messaging Toolkit When Writing Applications

To use the toolkit, include the ToolTalk Messaging Toolkit header file:

```
#include <Tt/tttk.h>
```

The ToolTalk Messaging Toolkit

This section contains a description of functions that are part of the ToolTalk Messaging Toolkit.

ttdt_close

```
Tt_status     ttdt_close(     const char * procid,
                              const char * new_procid,
                              int          sendStopped );
```

The `ttdt_close` function destroys a ToolTalk communication endpoint. This function calls the ToolTalk function `tt_close`.

- If the value of `procid` is `!= 0`, this function calls

  `tt_default_procid_set(  procid  )`
- If the value of `new_procid` is `!= 0`, this function calls

  `tt_default_procid_set(  new_procid )`
- If the `sendStopped` parameter is set, this function sends a Stopped notice.

The `ttdt_close` function can return any error returned by the ToolTalk functions `tt_default_procid_set` and `tt_close`. If the Sending notice fails, no errors are propagated.

ttdt_file_event

```
Tt_status    ttdt_file_event( Tt_message     context,
                              Tttk_op        event,
                              Tt_pattern *   patterns,
                              int            send );
```

The `ttdt_file_event` function uses the ToolTalk service to announce an event about a file. This function creates and, optionally, sends a ToolTalk notice that announces an event pertaining to a specified file. This file is indicated in the path name that was passed to the `ttdt_file_join` function when the `patterns` were created.

- Table A-3 describes the effect of the value of the *event* parameter on the announcement made.

Table A-3 Effect of event Parameter

Event Announced	Announcement
TTDT_MODIFIED	Registers in the scope passed to the ttdt_file_join function to announce the event to interested tools that handle Get_Modified, Save, and Revert requests.
TTDT_SAVED, TTDT_REVERTED	Unregisters handler patterns for Get_Modified, Save, and Revert requests. If the send parameter is set, this function sends a Saved or Reverted notice, respectively, in the scope.

- If the send parameter is set, this function sends the Modified notice in the scope.
- If the context parameter is a value other than zero, messages created by this routine inherit all contexts whose slotname begins with ENV_.

Table A-4 lists the possible errors that can be returned by this function.

Table A-4 Possible Errors Returned by ttdt_file_event

Error Returned	Description
TT_DESKTOP_EINVAL	The event notice was invalid. Valid event notices are TTDT_MODIFIED, TTD_TSAVED, and TTDT_REVERTED.
TT_ERR_POINTER	The patterns parameter was null.
TT_ERR_OVERFLOW	The ToolTalk service has received the maximum amount of active messages (2000) it can properly handle.
TT_ERR_NOMP	The ttsession process is not available. The ToolTalk service tries to restart ttsession if it is not running. This error indicates that the ToolTalk service is either not installed or not installed correctly.

ttdt_file_join

```
Tt_message  (*Ttdt_file_cb)(  Tt_message     message
                              Tttk_op        op,
                              char *         pathname,
                              void *         clientdata,
                              int            same_euid_egid,
                              int            same_procid );
Tt_pattern * ttdt_file_join(    const char *  pathname,
                              Tt_scope      the_scope,
                              int           join,
                              Ttdt_file_cb   cb,
                              void *         clientdata );
```

The `ttdt_file_join` function registers to observe ToolTalk events on the specified file. It registers in the scope to observe Deleted, Modified, Reverted, Moved, and Saved notices.

- The callback message argument `Ttdt_file_cb` takes the parameters listed in Table A-5.

Table A-5 Parameters taken by Ttdt_file_cb

Parameter	Description
`message`	The message being sent.
`op`	The operation being requested.
`pathname`	The path name of the file to which the message pertains. This copy can be freed with the ToolTalk function `tt_free`.
`clientdata`	The client data contained in the message.
`same_euid_egid`	A flag that identifies the sender; if this value is true, the sender can be trusted.
`same_procid`	A flag that identifies the sender; if this value is true, the sender is the same procid as the receiver.

- If the value of `the_scope` parameter is zero (that is, `TT_SCOPE_NONE`), the file scope is set to the default (`TT_BOTH`); however, if, for example, the ToolTalk database server `rpc.ttdbserverd` is not installed on the file server that owns `pathname`, the file scope is set to `TT_FILE_IN_SESSION`.

The ttdt_file_join function associates the value of the_scope and a copy of pathname with the Tt_patterns returned to allow the ttdt_file_quit function to access the patterns. The caller can modify or free pathname after the ttdt_file_join call returns.

- If the value of the join parameter is true, this function calls

```
tt_file_join(   pathname   )
```

This function returns a null-terminated array of Tt_pattern. Use the ttdt_file_quit function to destroy the array. If an error is returned, the returned array is an error pointer that can be decoded with tt_ptr_error. Table A-6 is a list of the possible errors returned by the ttdt_file_join function.

Table A-6 Possible Errors Returned by ttdt_file_join

Error Returned	Description
TT_ERR_NOMP	The ttsession process is not available. The ToolTalk service tries to restart ttsession if it is not running. This error indicates that the ToolTalk service is either not installed or not installed correctly.
TT_ERR_DBAVAIL	The ToolTalk service could not access the ToolTalk database needed for this operation.
TT_ERR_DBEXIST	The ToolTalk service did not find the specified ToolTalk database in the expected place.
TT_ERR_PATH	The ToolTalk service was not able to read a directory in the specified file path name.
TT_ERR_NOMEM	There is not enough memory available to perform the operation.

ttdt_file_notice

```
Tt_message   ttdt_file_notice(   Tt_message     context,
                                 Tttk_op        op,
                                 Tt_scope       scope,
                                 const char *   pathname,
                                 int           send_and_destroy );
```

The ttdt_file_notice function creates and, optionally, sends a standard ToolTalk notice about a file. Use this function to create the following standard file notices: Created, Deleted, Moved, Reverted, Saved, and Modified.

Note – The ttdt_file_event function is a higher-level interface than the ttdt_file_notice function and is the preferred method to send all notices except the Moved notice.

- If the context parameter is a value other than zero, messages created by this routine inherit all contexts whose slotname begins with ENV_.
- This function creates a notice with the specified *op* and *scope* parameters, and sets its file attribute to pathname parameter.
- If the send_and_destroy parameter is set, this function sends the message and then destroys it.

  If the value of the send_and_destroy parameter is false, the created message is returned; if the value of the send_and_destroy parameter is true, zero is returned.

If an error occurs, an error pointer is returned. Use tt_ptr_error to find out the Tt_status. Table A-7 describes possible errors returned by this function.

Table A-7 Possible Errors Returned by ttdt_file_notice

Error Returned	Description
TT_ERR_NOMP	The ttsession process is not available. The ToolTalk service tries to restart ttsession if it is not running. This error indicates that the ToolTalk service is either not installed or not installed correctly.
TT_ERR_PROCID	The process identifier specified is out of date or invalid.
TT_ERR_NOMEM	There is not enough memory available to perform the operation.
TT_ERR_OVERFLOW	The ToolTalk service has received the maximum amount of active messages (2000) it can properly handle.
TT_ERR_DBAVAIL	The ToolTalk service could not access the ToolTalk database needed for this operation.

Table A-7 Possible Errors Returned by ttdt_file_notice

Error Returned	Description
TT_ERR_DBEXIST	The ToolTalk service did not find the specified ToolTalk database in the expected place.
TT_DESKTOP_EINVAL	The operation was moved, and the value of the send_and_destroy parameter was true.
TT_ERR_POINTER	The path name was null, or was a ToolTalk error pointer.

ttdt_file_quit

```
Tt_status    ttdt_file_quit(Tt_pattern *  patterns,
                            int           quit );
```

The ttdt_file_quit function unregisters interest in ToolTalk events about a file. This function destroys patterns. If the quit parameter is set, this function calls

```
tt_file_quit( pathname )
```

Use this function to unregister interest in the path name that was passed to the ttdt_file_join function when patterns was created. Table A-8 lists the possible errors returned by this function.

Table A-8 Possible Errors Returned by ttdt_file_quit

Error Returned	Description
TT_ERR_NOMP	The ttsession process is not available. The ToolTalk service tries to restart ttsession if it is not running. This error indicates that the ToolTalk service is either not installed or not installed correctly.
TT_ERR_PROCID	The process identifier specified is out of date or invalid.

Table A-8 Possible Errors Returned by ttdt_file_quit

Error Returned	Description
TT_ERR_DBAVAIL	The ToolTalk service could not access the ToolTalk database needed for this operation.
TT_ERR_DBEXIST	The ToolTalk service did not find the specified ToolTalk database in the expected place.
TT_ERR_POINTER	The patterns were null or otherwise invalid.

ttdt_file_request

```
Tt_message        ttdt_file_request(
                         Tt_message              context,
                         Tttk_op                 op,
                         Tt_scope                scope,
                         const char              pathname,
                         Ttdt_file_cb            cb,
                         void                    client_data,
                         int                     send_and_destroy
                            );
```

The `ttdt_file_request` function creates, and optionally sends, any standard Desktop file-scoped request (such as Get_Modified, Save, and Revert).

Note – This function is a lower-level interface than the `ttdt_Get_Modified`, `ttdt_Save`, and `ttdt_Revert` functions, which create and send the request and then block on its reply.

The `ttdt_file_request` function creates a request with the specified `op` and `scope`, and sets its file attribute to `pathname`. Per Desktop messaging conventions, an unset Tt_mode argument of `TT_IN` and the vtype `File` is added to the request; and if the specified operation is `TTDT_GET_MODIFIED`, an unset Tt_mode argument of `TT_OUT` and the vtype `Boolean` is also added to the request.

If context is not zero, the request created by this routine inherits from context all contexts whose slotname are prefixed with ENV_.

This function installs cb as a message callback for the created request, and ensures that client data will be passed into the callback. If send is true, this function sends the request before returning the handle to it.

This function returns the created Tt_message when successful. If an error occurs, an error pointer is returned. Use tt_ptr_error to find out the Tt_status. Table A-8 lists the possible errors returned by this function.

Table A-9 Possible Errors Returned by ttdt_file_request

Error Returned	Description
TT_ERR_NOMP	The ttsession process is not available. The ToolTalk service tries to restart ttsession if it is not running. This error indicates that the ToolTalk service is either not installed or not installed correctly.
TT_ERR_PROCID	The process identifier specified is out of date or invalid.
TT_ERR_NOMEM	There is not enough available memory to perform the operation.
TT_ERR_OVERFLOW	The ToolTalk service has received the maximum amount of active messages (2000) it can properly handle.
TT_ERR_DBAVAIL	The ToolTalk service could not access the ToolTalk database needed for this operation.
TT_ERR_DBEXIST	The ToolTalk service did not find the specified ToolTalk database in the expected place.
TT_ERR_POINTER	The path name was null or otherwise invalid.

ttdt_Get_Modified

```
int          ttdt_Get_Modified(  Tt_message        context,
                                  const char *      pathname,
                                  Tt_scope          the_scope,
                                  XtAppContext      app2run,
                                  int               ms_timeout );
```

The ttdt_Get_Modified function asks if any ToolTalk client has changes pending on a file. This function sends a Get_Modified request and waits for a reply.

- If the context parameter is a value other than zero, messages created by this routine inherit all contexts whose slotname begins with ENV_.
- The Get_Modified request asks if any ToolTalk client has changes pending on pathname that it intends to make persistent.
- The the_scope parameter indicates the scope in which the Get_Modified request is sent. If the value of this parameter is zero (that is, TT_SCOPE_NONE), the file scope is set to the default (TT_BOTH); however, if, for example, the ToolTalk database server rpc.ttdbserverd is not installed on the file server that owns pathname, the file scope is set to TT_FILE_IN_SESSION.
- The app2run and ms_timeout parameters are passed to the tttk_block_while function to block on the reply to the Get_Modified request sent by this function.

If the Get_Modified request receives an affirmative reply within the specified time out, the ttdt_Get_Modified function returns non-zero; otherwise, it returns zero. This call does not return any errors.

ttdt_message_accept

```
Tt_pattern *   ttdt_message_accept(    Tt_message       contract,
                                       Ttdt_contract_cb cb,
                                       void *           clientdata,
                                       Widget           shell,
                                       int              accept,
                                       int              sendStatus
);
```

The ttdt_message_accept function accepts a contract to handle a ToolTalk request. A tool calls this function when it wants to accept responsibility for handling (that is, failing or rejecting) a request.

A Ttdt_contract_cb argument takes the parameters listed in Table A-10.

Table A-10 Parameters Taken by the Ttdt_contract_cb Argument

Parameter	Description
`Tt_message msg`	The request in the sent state. The client program must either fail, reject, or reply to the message.
`Tttk_op op`	The operation of the incoming request.
`Widget shell`	The shell passed to the `ttdt_message_accept` function.
`void *clientdata`	The client data passed to the `ttdt_message_accept` function.
`Tt_message contract`	The contract passed to the `ttdt_message_accept` function.

If the callback processes the message `msg` successfully, it returns zero; otherwise, it returns a `tt_error_pointer` cast to `Tt_message`.

If the callback does not consume the message `msg`, it returns the message and passes the `TT_CALLBACK_CONTINUE` routine down the call stack to offer the message to other callbacks, or to return it to the `tt_message_receive` call.

The `ttdt_message_accept` function registers in the default session for the handler-addressed requests described in Table A-11.

Table A-11 Requests for which ttdt_message_accept Registers

Request	How Request Is Handled
`Get_Geometry`, `Set_Geometry`	If the `shell` parameter is not null, these requests are handled transparently; if the `shell` parameter is null and the `cb` parameter is not null, these requests are passed to the callback routine; otherwise, these requests fail with the error `TT_DESKTOP_ENOTSUP`.
`Get_Iconified`, `Set_Iconified`	If the `shell` parameter is not null, these requests are handled transparently; if the `shell` parameter is null and the `cb` parameter is not null, these requests are passed to the callback routine; otherwise, these requests fail with the error `TT_DESKTOP_ENOTSUP`.
`Get_Mapped`, `Set_Mapped`	If the `shell` parameter is not null, these requests are handled transparently; if the `shell` parameter is null and the `cb` parameter is not null, these requests are passed to the callback routine; otherwise, these requests fail with the error `TT_DESKTOP_ENOTSUP`.
`Raise`	If the `shell` parameter is not null, this request is handled transparently; if the `shell` parameter is null and the `cb` parameter is not null, these requests are passed to the callback routine; otherwise, these requests fail with the error `TT_DESKTOP_ENOTSUP`.
`Lower`	If the `shell` parameter is not null, this request is handled transparently; if the `shell` parameter is null and the `cb` parameter is not null, these requests are passed to the callback routine; otherwise, these requests fail with the error `TT_DESKTOP_ENOTSUP`.
`Get_XInfo`, `Set_XInfo`	If the `shell` parameter is not null, these requests are handled transparently; if the `shell` parameter is null and the `cb` parameter is not null, these requests are passed to the callback routine; otherwise, these requests fail with the error `TT_DESKTOP_ENOTSUP`.
`Pause`	If the `cb` parameter is not null, this request is passed to the callback routine; otherwise, it fails with the error `TT_DESKTOP_ENOTSUP`.

Table A-11 Requests for which ttdt_message_accept Registers *(Continued)*

Request	How Request Is Handled
`Resume`	If the `cb` parameter is not null, this request is passed to the callback routine; otherwise, it fails with the error `TT_DESKTOP_ENOTSUP`.
`Quit`	If the `cb` parameter is not null, this request is passed to the callback routine; otherwise, it fails with the error `TT_DESKTOP_ENOTSUP`.
`Get_Status`	If the `cb` parameter is not null, this request is passed to the callback routine; otherwise, it fails with the error `TT_DESKTOP_ENOTSUP`.

If the `contract` argument has a `TT_WRN_START_MESSAGE` message status, the message caused the tool to be started.

Note – The started tool should join any scopes it wants to serve before accepting the contract so that it will receive any other messages already dispatched to its ptype; otherwise, the tool should undeclare its ptype while it is busy. If the tool does not join any scopes, the dispatched messages will cause other instances of the ptype to be started.

If the `accept` argument is true, the `ttdt_message_accept` function calls

```
tt_message_accept(   contract   )
```

If the `sendStatus` argument is true, the `ttdt_message_accept` function sends a Status notice to the requestor, using the parameters (if any) passed to the `ttdt_open` function.

This function returns a null-terminated array of `Tt_pattern`. Use the `tttk_patterns_destroy` function to destroy the array. If an error is returned, the returned array is an error pointer that can be decoded with `tt_ptr_error`. Table A-12 is a list of the possible errors returned by the `ttdt_message_accept` function.

Table A-12 Possible Errors Returned by ttdt_message_accept

Returned Error	Description
TT_ERR_NOMP	The ttsession process is not available. The ToolTalk service tries to restart ttsession if it is not running. This error indicates that the ToolTalk service is either not installed or not installed correctly.
TT_ERR_POINTER	The pointer passed does not point at an object of the correct type for this operation. For example, the pointer may point to an integer when a character string is needed.
TT_ERR_UNIMP	The ttsession for the default session is a version (1.0 or 1.0.1) that does not support the `tt_message_accept` function. **Note:** If the contract argument has a `TT_WRN_START_MESSAGE` message status, messages to the tool's ptype will remain blocked until the contract is rejected, replied to, or failed.

ttdt_open

```
char *        ttdt_open(   int *          ttfd,
                          const char *   toolname,
                          const char *   vendor,
                          const char *   version,
                          int            sendStarted );
```

The `ttdt_open` function creates a ToolTalk communication endpoint. This function calls `tt_open` and `tt_fd` functions. The `ttdt_open` function associates `toolname`, `vendor`, and `version` with the created procid. It initializes the new procid's default contexts from `environ(5)`. If the `sendStarted` argument is set, this function sends a Started notice.

The `ttdt_open` function returns the created procid in a string that can be freed with the `tt_free` function.

This function can return any error returned by the `tt_open` and `tt_fd` functions. If the Started notice fails, errors are not propagated.

ttdt_Revert

```
Tt_status    ttdt_Revert( Tt_message      context,
                          const char *    pathname,
                          Tt_scope        the_scope,
                          XtAppContext    app2run,
                          int             ms_timeout );
```

The `ttdt_Revert` function requests a ToolTalk client to revert a file. It sends a Revert request in `the_scope` and waits for a reply. The Revert request asks the handling ToolTalk client to discard any changes pending on `pathname`.

- If the `context` parameter is a value other than zero, messages created by this routine inherit all contexts whose slotname begins with `ENV_`.
- If the value of the `the_scope` parameter is zero (that is, `TT_SCOPE_NONE`), the file scope is set to the default (`TT_BOTH`); however, if, for example, the ToolTalk database server rpc.ttdbserverd is not installed on the file server that owns `pathname`, the file scope is set to `TT_FILE_IN_SESSION`.
- The `app2run` and `ms_timeout` parameters are passed to the `tttk_block_while` function to block on the reply to the Revert request sent by this function.

If the request receives an affirmative reply within the indicated timeout, the `ttdt_Revert` function returns `TT_OK`; otherwise, it returns either the `tt_message_status` of the failure reply, or one of the errors listed in Table A-13.

Table A-13 Possible Errors Returned by ttdt_Revert

Error Returned	Description
TT_ERR_NOMP	The ttsession process is not available. The ToolTalk service tries to restart ttsession if it is not running. This error indicates that the ToolTalk service is either not installed or not installed correctly.
TT_ERR_PROCID	The process identifier specified is out of date or invalid.
TT_ERR_NOMEM	There is not enough memory available to perform the operation.

Table A-13 Possible Errors Returned by ttdt_Revert *(Continued)*

Error Returned	Description
TT_ERR_OVERFLOW	The ToolTalk service has received the maximum amount of active messages (2000) it can properly handle.
TT_ERR_DBAVAIL	The ToolTalk service could not access the ToolTalk database needed for this operation.
TT_ERR_DBEXIST	The ToolTalk service did not find the specified ToolTalk database in the expected place.
TT_DESKTOP_ETIMEOUT	No reply was received before the allotted timeout.
TT_DESKTOP_EPROTO	The request was failed; however, the handler set the `tt_message_status` of the failure reply to `TT_OK` instead of a specific error status.
TT_ERR_POINTER	Path name was null, or was a ToolTalk error pointer.

ttdt_Save

```
Tt_status    ttdt_Save(Tt_message          context,
                       const char *        pathname,
                       Tt_scope            the_scope,
                       XtAppContext        app2run,
                       int                 ms_timeout );
```

The `ttdt_Save` function requests a ToolTalk client to save a file. It sends a Save request in `the_scope` and waits for a reply. The Save request asks the handling ToolTalk client to discard any changes pending on `pathname`.

- If the `context` parameter is a value other than zero, messages created by this routine inherit all contexts whose slotname begins with `ENV_`.
- If the value of the `the_scope` parameter is zero (that is, `TT_SCOPE_NONE`), the file scope is set to the default (`TT_BOTH`); however, if, for example, the ToolTalk database server `rpc.ttdbserverd` is not installed on the file server that owns `pathname`, the file scope is set to `TT_FILE_IN_SESSION`.

- The app2run and ms_timeout parameters are passed to the tttk_block_while function to block on the reply to the Save request sent by this function.

If the request receives an affirmative reply within the indicated timeout, the ttdt_Save function returns TT_OK; otherwise, it returns either the tt_message_status of the failure reply, or one of the errors listed in Table A-14.

Table A-14 Possible Returns of the ttdt_Save function

Error Returned	Description
TT_ERR_NOMP	The ttsession process is not available. The ToolTalk service tries to restart ttsession if it is not running. This error indicates that the ToolTalk service is either not installed or not installed correctly.
TT_ERR_PROCID	The process identifier specified is out of date or invalid.
TT_ERR_NOMEM	There is not enough memory available to perform the operation.
TT_ERR_OVERFLOW	The ToolTalk service has received the maximum amount of active messages (2000) it can properly handle.
TT_ERR_DBAVAIL	The ToolTalk service could not access the ToolTalk database needed for this operation.
TT_ERR_DBEXIST	The ToolTalk service did not find the specified ToolTalk database in the expected place.
TT_DESKTOP_ETIMEOUT	No reply was received before the allotted timeout.
TT_DESKTOP_EPROTO	The request was failed; however, the handler set the tt_message_status of the failure reply to TT_OK instead of a specific error status.
TT_ERR_POINTER	Path name was null, or was a ToolTalk error pointer.

ttdt_sender_imprint_on

```
Tt_status    ttdt_sender_imprint_on( const char *    handler,
                                      Tt_message      contract,
                                      char **         display,
                                      int *           width,
                                      int *           height,
                                      int *           xoffset,
                                      int *           yoffset,
                                      XtAppContext    app2run,
                                      int             ms_timeout );
```

The `ttdt_sender_imprint_on` function causes the calling tool ("ToolB") to adopt the behavior and certain characteristics of another tool ("ToolA"). ToolB adopts ToolA's X11 display, locale, and current working directory; it also learns ToolA's X11 geometry so that it can position itself appropriately.

If the `display` parameter is null, the environment variable `$DISPLAY` is set to ToolA's display; otherwise, ToolA's display is returned in this parameter. The returned value is a string that can be freed with the ToolTalk `tt_free` function.

This function sends a Get_Geometry request to ToolA. If ToolA does not return a value for any or all of the geometry parameters:

- If a value for the `width` parameter is not returned, it is set to `-1`.
- If a value for the `height` parameter is not returned, it is set to `-1`.
- If a value for the `xoffset` parameter is not returned, it is set to `INT_MAX`.
- If a value for the `yoffset` parameter is not returned, it is set to `INT_MAX`.

If the `width`, `height`, `xoffset`, and `yoffset` parameters in the `ttdt_sender_imprint_on` function are all set to null, a Get_Geometry request is not sent to ToolA.

The `app2run` and `ms_timeout` parameters are passed to the `tttk_block_while` function to block on the replies to the Get_Geometry request sent by this function.

Table A-15 lists the possible errors that can be returned by this function.

Table A-15 Possible Errors Returned by the ttdt_sender_imprint_on

Error Returned	Description
`TT_DESKTOP_ETIMEDOUT`	One or more of the sent requests did not complete before the allotted timeout.
`TT_ERR_NOMP`	The ttsession process is not available. The ToolTalk service tries to restart ttsession if it is not running. This error indicates that the ToolTalk service is either not installed or not installed correctly.
`TT_ERR_PROCID`	The process identifier specified is out of date or invalid.
`TT_ERR_NOMEM`	There is not enough memory available to perform the operation.
`TT_ERR_OVERFLOW`	The ToolTalk service has received the maximum amount of active messages (2000) it can properly handle.

ttdt_session_join

```
Tt_message   (*Ttdt_contract_cb)( Tt_message        msg,
                                  void *            clientdata
                                  Tt_message        contract );
Tt_pattern * ttdt_session_join(   const char *      sessid,
                                  Ttdt_session_cb   cb,
                                  Widget            shell,
                                  void *            clientdata,
                                  int               join );
```

The `ttdt_session_join` function joins a ToolTalk session as a "good desktop citizen"; that is, it registers patterns and default callbacks for many standard desktop message interfaces when it joins the session `sessid`. Table A-16 lists the message interfaces for which this function currently registers.

Table A-16 Standard Messages for which the ttdt_session_join Registers

Request	How Message Is Handled
Get_Environment, Set_Environment	These messages are handled transparently.
Get_Locale, Set_Locale	These messages are handled transparently.
Get_Situation, Set_Situation	These messages are handled transparently.
Signal	This message is handled transparently.
Get_Sysinfo	This message is handled transparently.
Get_Geometry, Set_Geometry	If the value of the shell parameter is not null and the shell is a realized *mappedWhenManaged applicationShellWidget*, these messages are handled transparently; if the shell is not a *mappedWhenManaged applicationShellWidget*, these messages fail with the error TT_DESKTOP_ENOTSUP.
Get_Iconified, Get_Iconified	If the value of the shell parameter is not null and the shell is a realized *mappedWhenManaged applicationShellWidget*, these messages are handled transparently; if the shell is not a *mappedWhenManaged applicationShellWidget*, these messages fail with the error TT_DESKTOP_ENOTSUP.
Get_Mapped, Set_Mapped	If the value of the shell parameter is not null and the shell is a realized *mappedWhenManaged applicationShellWidget*, these messages are handled transparently; if the shell is not a *mappedWhenManaged applicationShellWidget*, these messages fail with the error TT_DESKTOP_ENOTSUP.
Raise	If the value of the shell parameter is not null and the shell is a realized *mappedWhenManaged applicationShellWidget*, this message is handled transparently; if the shell is not a *mappedWhenManaged applicationShellWidget*, this message fails with the error TT_DESKTOP_ENOTSUP.

Table A-16 Standard Messages for which the ttdt_session_join Registers *(Continued)*

Request	How Message Is Handled
`Lower`	If the value of the `shell` parameter is not null and the shell is a realized *mappedWhenManaged applicationShellWidget*, this message is handled transparently; if the shell is not a *mappedWhenManaged applicationShellWidget*, this message fails with the error `TT_DESKTOP_ENOTSUP`.
`Get_XInfo`	If the value of the `shell` parameter is not null, this message is handled transparently; otherwise, this message fails with the error `TT_DESKTOP_ENOTSUP`.
`Set_XInfo`	If the value of the `shell` parameter is not null and the shell is a realized *mappedWhenManaged applicationShellWidget*, this message is handled transparently; if the shell is not a *mappedWhenManaged applicationShellWidget*, this message fails with the error `TT_DESKTOP_ENOTSUP`.
`Pause`	If the `cb` parameter is not null, this message is passed to the callback; the cb parameter is null, this message fails with the error `TT_DESKTOP_ENOTSUP`.
`Resume`	If the `cb` parameter is not null, this message is passed to the callback; the `cb` parameter is null, this message fails with the error `TT_DESKTOP_ENOTSUP`.
`Quit`	If the `cb` parameter is not null, this message is passed to the callback; the `cb` parameter is null, this message fails with the error `TT_DESKTOP_ENOTSUP`.
`Get_Status`	If the `cb` parameter is not null, this message is passed to the callback; the `cb` parameter is null, this message fails with the error `TT_DESKTOP_ENOTSUP`.
`Do_Command`	If the `cb` parameter is not null, this message is passed to the callback; the `cb` parameter is null, this message fails with the error `TT_DESKTOP_ENOTSUP`.

If the sessid parameter is null, the default session is joined.

If the join parameter is set, the specified session is joined.

A Ttdt_contract_cb message takes the parameters described in Table A-17. If the callback does not consume the message, it returns the message; if it consumes the message, it returns either zero or a error pointer cast to Tt_message.

Table A-17 Parameters taken by Ttdt_session_cb

Parameter	Description
Tt_message msg	The request in the sent state. The client program must either fail, reject, or reply to the message. **Note**: Destroy the message *msg* after it is processed.
void *clientdata	The clientdata passed to either the ttdt_session_join or ttdt_message_accept function.
Tt_message contract	The contract passed to the ttdt_message_accept function. If the callback is installed by the ttdt_session_join function, the value for the contract parameter is always zero.

The ttdt_session_join function returns a null-terminated array of Tt_pattern, which can be passed to the ttdt_session_quit function to be destroyed. If an error occurs, the returned array that is an error pointer. Use tt_ptr_error to find the Tt_status. Table A-18 lists the possible errors returned.

Table A-18 Possible Errors Returned by the ttdt_session_join

Error Returned	Description
TT_ERR_NOMP	The ttsession process is not available. The ToolTalk service tries to restart ttsession if it is not running. This error indicates that the ToolTalk service is either not installed or not installed correctly.
TT_ERR_PROCID	The process identifier specified is out of date or invalid.
TT_ERR_SESSION	An out-of-date or invalid ToolTalk session was specified.

Table A-18 Possible Errors Returned by the ttdt_session_join *(Continued)*

Error Returned	Description
`TT_ERR_POINTER`	The pointer passed does not point at an object of the correct type for this operation. For example, the pointer may point to an integer when a character string is needed.
`TT_ERR_NOMEM`	There is not enough memory available to perform the operation.

ttdt_session_quit

```
Tt_status    ttdt_session_quit( const char *      sessid,
                                Tt_pattern *      sess_pats,
                                int               quit );
```

The `ttdt_session_quit` function quits a ToolTalk session as a "good desktop citizen"; that is, it unregisters all the patterns and default callback it registered when it joined the session.

This function destroys all patterns in `sess_pats`. If the `quit` parameter is set, it quits the session `sessid`; if the `sessid` parameter is null, it quits the default session.

Table A-19 lists the errors that can be returned by this function.

Table A-19 Possible Errors Returned by the ttdt_session_quit

Error Returned	Description
`TT_ERR_NOMP`	The ttsession process is not available. The ToolTalk service tries to restart ttsession if it is not running. This error indicates that the ToolTalk service is either not installed or not installed correctly.

Table A-19 Possible Errors Returned by the ttdt_session_quit

Error Returned	Description
TT_ERR_PROCID	The process identifier specified is out of date or invalid.
TT_ERR_SESSION	An out-of-date or invalid ToolTalk session was specified.
TT_ERR_POINTER	The pointer passed does not point at an object of the correct type for this operation. For example, the pointer may point to an integer when a character string is needed.

ttdt_subcontract_manage

```
Tt_pattern * ttdt_subcontract_manage(  Tt_message     subcontract,
                                      Ttdt_contract_cb cb,
                                      Widget           shell,
                                      void *           clientdata );
```

The `ttdt_subcontract_manage` function manages an outstanding request. It allows the requesting tool to manage the standard Desktop interactions with the tool that is handling the request. This function registers in the default session for `TT_HANDLER`-addressed Get_Geometry and Get_XInfo requests, and Status notices.

If the `shell` parameter is null, the request or notice is passed to the `cb` parameter; otherwise, the request is handled transparently.

The `ttdt_subcontract_manage` function returns a null-terminated array of Tt_pattern, which can be passed to the `ttdt_session_quit` function to be destroyed. If an error occurs, the returned array that is an error pointer. Use `tt_ptr_error` to find the `Tt_status`. Table A-20 lists the possible errors returned.

Table A-20 Possible Errors Returned by the ttdt_subcontract_manage

Error Returned	Description
TT_ERR_NOMEM	There is not enough memory available to perform the operation.
TT_ERR_NOMP	The ttsession process is not available. The ToolTalk service tries to restart ttsession if it is not running. This error indicates that the ToolTalk service is either not installed or not installed correctly.
TT_ERR_PROCID	The process identifier specified is out of date or invalid.
TT_ERR_POINTER	The `subcontract` parameter was not a valid `Tt_message`.
TT_ERR_EINVAL	Both the `shell` and `cb` parameters were null.

ttmedia_Deposit

```
Tt_status  ttmedia_Deposit(   Tt_message               load_contract,
                              const char *             buffer_id,
                              const char *             media_type,
                              const unsigned char *    new_contents,
                              int                      new_len,
                              const char *             pathname,
                              XtAppContext             app2run,
                              int                      ms_timeout );
```

The `ttmedia_Deposit function` sends a Deposit request to checkpoint a document that was the subject of a Media Exchange `load_contract` request such as Edit, Compose, or Open.

This function creates and sends a Deposit request and returns the success or failure of that request.

- `load_contract` is the request that caused this editor to load the document.
- `buffer_id` is the id of the buffer this editor created if the document was loaded by an Open request.
- `media_type` is the vtype of the `contents` argument of the sent request.
- `new_contents` and `new_len` are the values for the `contents` argument.

After the request is sent, `app2run` and `ms_timeout` are passed to the `tttk_block_while` function to wait for the reply.

Table A-21 Possible Errors Returned by the ttmedia_Deposit

TT_ERR_NOMP	The ttsession process is not available. The ToolTalk service tries to restart ttsession if it is not running. This error indicates that the ToolTalk service is either not installed or not installed correctly.
TT_ERR_PROCID	The process identifier specified is out of date or invalid.
TT_ERR_NOMEM	There is not enough available memory to perform the operation.
TT_ERR_OVERFLOW	The ToolTalk service has received the maximum amount of active messages (2000) it can properly handle.
TT_ERR_DBAVAIL	The ToolTalk service could not access the ToolTalk database needed for this operation.
TT_ERR_DBEXIST	The ToolTalk service did not find the specified ToolTalk database in the expected place.
TT_DESKTOP_ETIMEOUT	No reply was received before the allotted timeout.
TT_ERR_POINTER	Path name was null, or was a ToolTalk error pointer.

ttmedia_load

```
Tt_message  (*Ttmedia_load_msg_cb)(Tt_message          msg,
                                   void *              clientdata,
                                   Tttk_op             op,
                                   unsigned char *     contents,
                                   int                 len,
                                   char *              file  );
Tt_message   ttmedia_load(         Tt_message          context,
                                   Ttmedia_load_msg_cb cb,
                                   void *              clientdata,
                                   Tttk_op             op,
                                   const char *        media_type,
                                   const unsigned char* contents,
                                   int                 len,
```

```
const char *        file,
const char *        docname,
int                 send );
```

The ttmedia_load function creates and, optionally, sends a Media Exchange request to display, edit, or compose a document. This function creates and sends Display, Edit, or Compose requests. It returns a subcontract with the tool managing the request.

Note – Use the ttdt_subcontract_manage function immediately after sending the request created by this message to manage the standard interactions with the handler of the request.

If value of the context argument is not zero, messages created by this routine inherit all contexts whose slotname begins with ENV_.

The clientdata argument is passed to the cb argument when the reply is received, or when intermediate versions of the document are checkpointed through Deposit requests.

The op argument must be either TTME_DISPLAY, TTME_EDIT, or TTME_COMPOSE.

The media_type argument names the data format of the document. This argument usually determines which application is chosen to handle the request.

The contents and len arguments specify the document. If the value of both of these arguments is zero and the value of the file argument is not zero, the document is assumed to be contained in the specified file.

If the docname argument is not null, it is used as the title of the document.

If the send argument is true, the message is sent before it is returned.

Table A-22 lists the parameters taken by a Ttmedia_load_msg_cb message.

Table A-22 Parameters Taken by the Ttmedia_load_msg_cb

Parameter	Description
`Tt_message msg`	The reply to the request, or a Deposit request with a `messageID` argument that names the `tt_message_id` of the load request. If the value of this parameter is a Deposit request, the client program must either fail or reply to the request. **Note**: Destroy the message msg after it is processed.
`Tttk_op op`	The operation of the message (either `TTME_DEPOSIT` or the operation passed to the `ttmedia_load` message).

`unsigned char * contents`
`int len`
`char *file`
	The contents of the arriving document. If the `len` argument is zero, the document is contained in the specified file. If the `contents` or `file` arguments are non-null, use the ToolTalk function `tt_free` to free them.

`void *clientdata`	The client data passed to the `ttmedia_load` message.

If the message is processed successfully, the callback returns zero; if the processing results in an error, the callback returns an error pointer cast to `Tt_message`.

If the callback does not consume the message `msg`, it returns the message and the toolkit passes the `TT_CALLBACK_CONTINUE` routine down the call stack to offer the message to other callbacks, or to return it to the `tt_message_receive` call.

Upon completion, the `ttmedia_load` function returns the request it was asked to build. If an error occurs, this function returns an error pointer. Use `tt_ptr_error` to find the `Tt_status`. Table A-23 lists the possible errors returned.

Table A-23 Possible Errors Returned by the ttmedia_load

Error Returned	Description
TT_ERR_NOMP	The ttsession process is not available. The ToolTalk service tries to restart ttsession if it is not running. This error indicates that the ToolTalk service is either not installed or not installed correctly.
TT_ERR_PROCID	The process identifier specified is out of date or invalid.
TT_ERR_NOMEM	There is not enough memory available to perform the operation.
TT_ERR_OVERFLOW	The ToolTalk service has received the maximum amount of active messages (2000) it can properly handle.

ttmedia_load_reply

```
Tt_message  ttmedia_load_reply( Tt_message           contract,
                                const unsigned char *new_contents,
                                int                  new_len,
                                int                  reply_and_destroy );
```

Use the `ttmedia_load_reply` function to reply to a Media Exchange request to display, edit, or compose a document.

If both the `new_contents` and `new_len` arguments are non-zero, their value is used to set the new contents of the document in the appropriate output argument of the `contract` argument. If the `reply_and_destroy` argument is true, a reply is made to the `contract` argument and then the message is destroyed.

Table A-24 lists the possible errors returned.

Table A-24 Possible Errors Returned by the ttmedia_load_reply

Error Returned	Description
TT_ERR_NOMP	The ttsession process is not available. The ToolTalk service tries to restart ttsession if it is not running. This error indicates that the ToolTalk service is either not installed or not installed correctly.
TT_ERR_PROCID	The process identifier specified is out of date or invalid.
TT_ERR_NUM	
TT_ERR_NOTHANDLER	

ttmedia_ptype_declare

```
Tt_message (*Ttmedia_load_pat_cb)(  Tt_message           msg,
                                     void *               clientdata,
                                     Tttk_op              op,
                                     Tt_status            diagnosis,
                                     unsigned char *      contents,
                                     int                  len,
                                     char *               file,
                                     char *               docname );
Tt_status  ttmedia_ptype_declare(    const char *         ptype,
                                     int                  base_opnum,
                                     Ttmedia_load_pat_cb  cb,
                                     void *               clientdata,
                                     int                  declare );
```

The `ttmedia_ptype_declare` function declares the ptype of a Media Exchange media editor. This function initializes an editor that implements the Media Exchange message interface for a particular media type.

- It calls the `cb` argument when the editor is asked to edit a document of the kind supported by `ptype`.

- It installs a toolkit-internal operation number (opnum) callback on a series of signatures that the ptype is assumed to contain. The toolkit-internal opnum callback passes clientdata to the cb argument when a request is received that matches one of these signatures. The opnums start at base_opnum, which must be zero or a multiple of 1000.
- If the declare argument is true, it calls

```
tt_ptype_declare(   ptype  )
```

If the ptype implements several different media types, the ttmedia_ptype_declare function can be called multiple times. Each call must have a different base_opnum value.

Note – The ttmedia_ptype_declare function can be called multiple times; however, the declare argument can "true" only once for each type.

Table A-25 lists the parameters taken by a Ttmedia_load_pat_cb message.

Table A-25 Parameters Taken by Ttmedia_load_pat_cb

Parameter	Description
Tt_message msg	The request sent. The client program must either fail, reject, or reply to the request.
Tttk_op op	The operation of the incoming request (either TTME_COMPOSE, TTME_EDIT, or TTME_DISPLAY.)
Tt_status diagnosis	The error code with which the toolkit recommends the request should be failed (for example, TT_DESKTOP_ENODATA). If the diagnosis is not TT_OK and the callback routine returns the message msg, the toolkit fails the message msg and destroys it.
unsigned char * contents int len char *file	The contents of the arriving document. If the len argument is zero, the document is contained in specified file. If value of the contents or file arguments is non-null, use the ToolTalk function tt_free to free them.
char * docname	The name of the document, if any.
void * clientdata	The client data passed to the ttmedia_ptype_declare message.

If the message is processed successfully, the callback returns zero; if the processing results in an error, the callback returns an error pointer cast to `Tt_message`.

If the callback does not consume the message `msg` and the value of the `diagnosis` argument is not `TT_OK`, it returns the message and the toolkit passes the `TT_CALLBACK_CONTINUE` routine down the call stack to offer the message to other callbacks, or to return it to the `tt_message_receive` call.

If an error occurs, this function returns one of the errors listed in Table A-26.

Table A-26 Possible Errors Returned by the ttmedia_ptype_declare

Error Returned	Description
TT_ERR_NOMP	The ttsession process is not available. The ToolTalk service tries to restart ttsession if it is not running. This error indicates that the ToolTalk service is either not installed or not installed correctly.
TT_ERR_PROCID	The process identifier specified is out of date or invalid.
TT_ERR_PTYPE	The ToolTalk service could not locate the specified ptype.
TT_ERR_POINTER	The pointer passed does not point at an object of the correct type for this operation. For example, the pointer may point to an integer when a character string is needed.

tttk_block_while

```
Tt_status      tttk_block_while(
                              const int       *blocked,
                              int              ms_timeout );
```

The `tttk_block_while` function blocks the program while it awaits a reply for the `ms_timout` time.

tttk_message_abandon

```
Tt_status  tttk_message_abandon( Tt_messagemsg );
```

The `tttk_message_abandon` function abandons the request, and then destroys it.

Note – A program should abandon a message when it does not understand the message and wants to dispose of it.

If an error occurs, this function returns one of the errors listed in Table A-27.

Table A-27 Possible Errors Returned by the tttk_message_abandon

Error Returned	Description
TT_ERR_NOMP	The ttsession process is not available. The ToolTalk service tries to restart ttsession if it is not running. This error indicates that the ToolTalk service is either not installed or not installed correctly.
TT_ERR_POINTER	The pointer passed does not point at an object of the correct type for this operation. For example, the pointer may point to an integer when a character string is needed.
TT_ERR_NOTHANDLER	

tttk_message_create

```
Tt_message   tttk_message_create(Tt_message           context,
                                  Tt_class             the_class,
                                  Tt_scope             the_scope,
                                  const char *         handler,
                                  const char *         op,
                                  Tt_message_callback callback );
```

The `tttk_message_create` function creates a message that conforms to the conventions. This function provides a simple way to create a message that propagates inherited contexts from one message to another.

The `tttk_message_create` function creates a message and copies onto it all the context slots from `context` whose slotname begins with `ENV_`. The created message is given a `Tt_class` value of `the_class` and a `Tt_scope` value of `the_scope`.

If the handler parameter is null, the message is given a Tt_address of TT_PROCEDURE; otherwise, the message is TT_HANDLER-addressed to that procid.

If the op argument is not null, the message's op argument is set to that value.

If the callback argument is not null, it is added to the message as a message callback.

If successful, the tttk_message_create function returns the created Tt_message, which can be modified, sent, and destroyed in the same way as any other Tt_message.

If an error occurs, an error pointer is returned. Use tt_ptr_error to find the Tt_status. Table A-28 lists the possible errors returned.

Table A-28 Possible Errors Returned by the tttk_message_create

Error Returned	Description
TT_ERR_NOMP	The ttsession process is not available. The ToolTalk service tries to restart ttsession if it is not running. This error indicates that the ToolTalk service is either not installed or not installed correctly.
TT_ERR_PROCID	The process identifier specified is out of date or invalid.
TT_ERR_NOMEM	There is not enough memory available to perform the operation.

tttk_message_destroy

```
Tt_status   tttk_message_destroy(Tt_message    msg );
```

The tttk_message_destroy function destroys any message that conforms to the conventions.

Note – This message can be used in place of the tt_message_destroy message.

The tttk_message_destroy function destroys any patterns that may have been stored on the message by the ttdt_message_accept or ttdt_subcontract_manage functions and then passes the message msg to the tt_message_destroy function.

This function returns the value returned by the tt_message_destroy function.

tttk_message_fail

```
Tt_status   tttk_message_fail(
                                  Tt_message       msg,
                                  Tt_status        status,
                                  const char       *status_string,
                                  int              destroy
              );
```

The tttk_mesage_fail function fails the message msg and then destroys it.

Note – A program should abandon a message when it does not understand the message and wants to dispose of it.

A message whose state is TT_SENT can be failed. If the message is a handler-addressed message, or if it has a tt_message_status of TT_WRN_START_MESSAGE, it can be failed.

This function returns TT_DESKTOP_ENOTSUP.

tttk_message_receive

```
Tt_status      tttk_message_receive( const char*procid );
```

The tttk_message_receive function calls the tt_message_receive function to retrieve the next ToolTalk message.

If procid != 0, this function calls

tt_default_procid_set(*procid*)

tttk_message_reject

```
Tt_status  tttk_message_reject(
                                Tt_message          msg,
                                Tt_status           status,
                                const char*         status_string,
                                int                 destroy};
```

The `tttk_message_reject` function rejects the message `msg` and then destroys it.

Note – A program should abandon a message when it does not understand the message and wants to dispose of it.

A message whose state is `TT_SENT` can be rejected. If the message is `not` a handler-addressed message, or if it has a `tt_message_status` other than `TT_WRN_START_MESSAGE`, it can be rejected.

This function returns `TT_DESKTOP_ENOTSUP`.

tttk_op_string

```
char          *tttk_op_string(Tttk_opop);
```

The `tttk_op_string` function returns string for the operation `op` if successful; otherwise, this function returns zero.

Note – Use the `tt_free` function to free the string returned.

```
Tttk_op    tttk_string_op(    const char *opstring );
```

The `tttk_string_op` function returns a string containing the operation for the specified string. On error, this function returns `TTDT_OP_NONE`.

tttk_Xt_input_handler

```
void       tttk_Xt_input_handler(    XtPointer     procid,
                                    int *          source,
                                    XtInputId *    id );
```

The tttk_Xt_input_handler function processes ToolTalk events for Xt clients. Use this function as your Xt input handler unless you expect some messages not to be consumed by callbacks.

This function passes the procid argument to the tttk_message_receive function and passes any returned message (that is, messages that are not consumed by callbacks) to the tttk_message_abandon function.

If this function returns the error TT_ERR_NOMP, the tttk_Xt_input_handler function will pass the id parameter to the XtRemoveInput function.

The CoEd Demonstration Program B

This appendix contains the files and source code listing showing the ToolTalk related code for a ToolTalk demonstration program called CoEd. The CoEd demo program uses the ToolTalk Desktop Services message set. It illustrates how an editor can use the ToolTalk service to keep all changes made by the user in sync if multiple instances of the editor are editing the same file at the same time.

The CoEd Ptype File

The CoEd ptype file, shown in the example below.

```
ptype DT_CoEd {                                           /* Process type identifier */
    start "CoEd";                                         /* Start string */
    handle:                                               /* Receiving process */
    /*
     * Display ISO_Latin_1
     */
    session        Display( in    ISO_Latin_1 contents) => start opnum = 1; /* Signature */
    session        Display( in    ISO_Latin_1 contents,
                                   in    messageID   counterfoil) => start opnum = 2;
    session        Display( in    ISO_Latin_1 contents,
                                   in    title       docName) => start opnum = 3;
    session        Display( in    ISO_Latin_1 contents,
                                   in    messageID   counterfoil,
                                   in    title       docName) => start opnum = 4;
    /*
     * Edit ISO_Latin_1
```

```
     */
    session             Edit(   inout ISO_Latin_1 contents) => start opnum = 101;
    session             Edit(       inout ISO_Latin_1 contents,
                                    in    messageID   counterfoil) => start opnum = 102;
    session             Edit(       inout ISO_Latin_1 contents,
                                    in    title       docName) => start opnum = 103;
    session             Edit(       inout ISO_Latin_1 contents,
                                    in    messageID   counterfoil,
                                    in    title       docName) => start opnum = 104;
    /*
     * Compose ISO_Latin_1
     */
    session             Edit(       out   ISO_Latin_1 contents) => start opnum = 201;
    session             Edit(       out   ISO_Latin_1 contents,
                                    in    messageID   counterfoil) => start opnum = 202;
    session             Edit(       out   ISO_Latin_1 contents,
                                    in    title       docName) => start opnum = 203;
    session             Edit(       out   ISO_Latin_1 contents,
                                    in    messageID   counterfoil,
                                    in    title       docName) => start opnum = 204;
    /*
     * Open an ISO_Latin_1 buffer
     */
    session             Open(           in    ISO_Latin_1 contents,
                                    out   bufferID    docBuf,
                                    in    boolean     readOnly       ) => start opnum = 400;
    session             Open(       in    ISO_Latin_1 contents,
                                    out   bufferID    docBuf,
                                    in    boolean     readOnly,
                                    in    boolean     mapped         ) => start opnum = 401;
    session             Open(       in    ISO_Latin_1 contents,
                                    out   bufferID    docBuf,
                                    in    boolean     readOnly,
                                    in    boolean     mapped,
                                    in    integer     shareLevel     ) => start opnum = 402;
    session         Open(       in    ISO_Latin_1 contents,
                                    out   bufferID    docBuf,
                                    in    boolean     readOnly,
                                    in    boolean     mapped,
                                    in    integer     shareLevel,
                                    in    locator     initialPos     ) => start opnum = 403;
};
```

The CoEd.C File

The CoEd.C file, shown in Code Example B-2, shows the ToolTalk code that needs to be included in every application to initialize the toolkit, join a ToolTalk session and registering patterns, and add the ToolTalk service to its event loop.

Note – This file also contains ToolTalk code that is specific to CoEd in its role as an editor application. This code includes declaring a ptype and processing the start message.

```
/*
 * CoEd.cc
 *
 * Copyright (c) 1991,1993 by Sun Microsystems.
 */

#include <stdlib.h>
#include <desktop/tttk.h>                       // Include the ToolTalk messaging toolkit
#include <CoEd.h>
#include "CoEditor.h"
#include "CoEdTextBuffer.h"

XtAppContext          myContext;
Widget                myTopWidget        = 0;
Display       *myDpy;
int         abortCode= 0;
Tt_pattern     *sessPats= 0;                             // Patterns returned when session
joined
int                timeOutFactor         = 1000;
int             maxBuffers               = 1000;
int          *pArgc;
char       **globalArgv;

const char     *ToolName= "CoEd";
const char     *usage =
"Usage: CoEd [-p01] [-w n] [-t n] [file]\n"
"   -p      print ToolTalk procid\n"
"   -0      do not open an initial composition window\n"
"   -1      be a single-buffer editor\n"
"   -w      sleep for n seconds before coming up\n"
"   -t      use n as timeout factor, in milliseconds (default: 1000)\n"
;
```

```
void
main(
    int    argc,
    char **argv
)
{
    static const char *here = "main()";
    int   delay   = 0;
    int   printid = 0;
    int   compose = 1;
    char *file    = 0;

    OlToolkitInitialize( 0 );
    XtToolkitInitialize();
    myContext = XtCreateApplicationContext();
    //
    // This display may get closed, and another opened, inside
    // CoEditor::_init(), if e.g. our parent is on a different screen
    //
    pArgc = &argc;
    globalArgv = argv;
    myDpy = XtOpenDisplay( myContext, 0, 0, "CoEd", 0, 0, &argc, argv );

    int c;
    while ((c = getopt( argc, argv, "p01w:t:" )) != -1) {
            switch (c) {
                case 'p':
            printid = 1;
            break;
            case '0':
            compose = 0;
            break;
            case '1':
            maxBuffers = 1;
            break;
            case 'w':
            delay = atoi( optarg );
            break;
            case 't':
            timeOutFactor = atoi( optarg );
            break;
            default:
            fputs( usage, stderr );
            exit( 1 );
            }
```

```
}
if (optind < argc) {
        file = argv[ optind ];
}
while (delay > 0) {
        sleep( 1 );
        delay--;
}

int myTtFd;                                              // Obtain process identifier
// Initialize toolkit and create a ToolTalk communication endpoint
char *myProcID = ttdt_open( &myTtFd, ToolName, "SunSoft", "%I", 1 );

// Declare ptype
ttmedia_ptype_declare( "DT_CoEd", 0, CoEditor::loadISOLatin1_,
               (void *)&myTopWidget, 1 );

// Process the message that started us, if any
tttk_Xt_input_handler( 0, 0, 0 );
if (abortCode != 0) {
        // Error in message that caused us to start.
        exit( abortCode );
}

if (CoEditor::numEditors == 0) {
        // started by hand, not by ToolTalk
        if (file == 0) {
        if (compose) {
            new CoEditor( &myTopWidget );
        }
        } else {
            new CoEditor( &myTopWidget, file );
        }
}
//
// If sessPats is unset, then we have not joined the desktop
// session yet.  So join it.
//
if (sessPats == 0) {
        Widget session_shell = CoEditor::editors[0]->shell;
        if (maxBuffers > 1) {
            //
            // In multi-window mode, no single window is the
            // distinguished window.
            //
            session_shell = myTopWidget;
```

```
            }
            sessPats = ttdt_session_join( 0, 0, session_shell, 0, 1 );
    }

        XtAppAddInput( myContext, myTtFd, (XtPointer)XtInputReadMask,
                tttk_Xt_input_handler, myProcID );

    XtAppMainLoop( myContext );
}
```

The Coeditor.C File

The `Coeditor.C` file, shown in the example below, shows the ToolTalk code that needs to be included in every editor application to pass a media callback and reply when a request has been completed. It also shows other optional ToolTalk functions that can be included in an editor application.

Note – Ellipses (...) indicates code that has been omitted.

```
...

CoEditor::CoEditor(
    Widget *parent
)
{
    _init();
    _init( parent );
}

CoEditor::CoEditor(
    Widget     *parent,
    const char *file
)
{
    _init();
    _init( parent );
    _load( file );
}

CoEditor::CoEditor(
    Widget       *parent,
    Tt_messagemsg,
```

```
    const char      * /*docname*/,
    Tt_status       &status
)
{
    _init();
    status = _init( msg );
    if (status != TT_OK) {
            return;
    }
    _init( parent );
    status = _acceptContract( msg );
}

CoEditor::CoEditor(
    Widget       *parent,
    Tt_message        msg,
    int                        /*readOnly*/,
    const char      *file,
    const char      * /*docname*/,
    Tt_status       &status
)
{
    _init();
    status = _init( msg );
    if (status != TT_OK) {
            return;
    }
    _init( parent );
    status = _load( file );
    if (status != TT_OK) {
            return;
    }
    status = _acceptContract( msg );
}

CoEditor::CoEditor(
    Widget       *parent,
    Tt_messagemsg,
    int                    /*readOnly*/,
    unsigned char  *contents,
    int             /*len*/,
    const char      * /*docname*/,
    Tt_status       &status
)
{
    _init();
```

```
    status = _init( msg );
    if (status != TT_OK) {
            return;
    }
    _init( parent );
    XtVaSetValues( (Widget)_text,
            XtNsourceType,         (XtArgVal)OL_STRING_SOURCE,
            XtNsource,             (XtArgVal)contents,
            NULL );
    _textBuf  = OlTextEditTextBuffer( _text );
    RegisterTextBufferUpdate( _textBuf, CoEditor::_textUpdateCB_,
                (caddr_t)this );
    status = _acceptContract( msg );
}

CoEditor::~CoEditor()
{
    //
    // No need for a separate save if we are sending the document
    // back in a reply.
    //
    if (_contract == 0) {
            if (_modifiedByMe) {
            // we revert before quitting if we don't want to save
            _save();
            }
    } else {
            int   len;
            char *contents = _contents( &len );
            // Reply to media load callback with edited contents of text
            ttmedia_load_reply( _contract, (unsigned char *)contents,
                    len, 1 );
            if (contents != 0) {
            XtFree( contents );
            }
        _contract = 0;
    }
    numEditors--; // XXX assumes user destroys windows LIFO!
}

Tt_message
CoEditor::loadISOLatin1_(
    Tt_message            msg,
    Tttk_op            op,
    Tt_status          diagnosis,
    unsigned char  *contents,
```

```
    int             len,
    char       *file,
    char       *docname,
    void           *pWidget
)
{
    static const char *here = "CoEditor::loadISOLatin1_()";

    Tt_status status   = TT_OK;
    CoEditor *coEditor = 0;
    if (diagnosis != TT_OK) {
            // toolkit detected an error
            if (tt_message_status( msg ) == TT_WRN_START_MESSAGE) {
            //
            // Error is in start message!  We now have no
            // reason to live, so tell main() to exit().
            //
            abortCode = 2;
            }
            // let toolkit handle the error
            return msg;
    }
    if ((op == TTME_COMPOSE) && (file == 0)) {
            coEditor = new CoEditor( (Widget *)pWidget, msg, docname,
                     status );
    } else if (len > 0) {
            coEditor = new CoEditor( (Widget *)pWidget, msg,
                     (op == TTME_DISPLAY),
                     contents, len, docname, status );
    } else if (file != 0) {
            coEditor = new CoEditor( (Widget *)pWidget, msg,
                     (op == TTME_DISPLAY),
                     file, docname, status );
    } else {
            // Fail a message
            tttk_message_fail( msg, TT_DESKTOP_ENODATA, 0, 1 );
    }
    tt_free( (caddr_t)contents );
    tt_free( file );
    tt_free( docname );
    return 0;
}

void
CoEditor::_init()
{
```

```
    _baseFrame         = 0;
    _controls          = 0;
    _fileBut           = 0;
    _editBut           = 0;
    _scrolledWin= 0;
    _text              = 0;
    _textBuf           = 0;
    _modifiedByMe= FALSE;
    _modifiedByOther= 0;
    _contract          = 0;
    _contractPats= 0;
    _filePats          = 0;
    _file              = 0;
    _x      = INT_MAX;
    _y      = INT_MAX;
    _w      = INT_MAX;
    _h      = INT_MAX;
}

Tt_status
CoEditor::_init(
    Tt_message msg
)
{
    int width, height, xOffset, yOffset;
    width = height = xOffset = yOffset = INT_MAX;
    _contract = msg;
    ttdt_sender_imprint_on( 0, msg, 0, &_w, &_h, &_x, &_y,
               10 * timeOutFactor );
    return TT_OK;
}

typedef enum {
    Open,
    Save,
    SaveAs,
    Revert
} FileOp;

static const char *fileButs[] = {
    "Open...",
    "Save",
    "Save as...",
    "Revert"
};
```

```
const int numFileButs = sizeof( fileButs ) / sizeof( const char * );

typedef enum {
    Undo,
    Cut,
    Copy,
    Paste,
    Delete,
    SelText,
    SelAppt
} EditOp;

static const char *editButs[] = {
    “Undo”,
    “Cut”,
    “Copy”,
    “Paste”,
    “Delete”,
    “Text as ISO_Latin_1”,
    “Text as Appointment”
};

const int numEditButs = sizeof( editButs ) / sizeof( const char * );

void
CoEditor::_init(
    Widget *parent
)
{
    if (*parent == 0) {
            if (_contract != 0) {
            //
            // Re-open display, since $DISPLAY may have changed by
            // ttdt_sender_imprint_on().
            //
            XtCloseDisplay( myDpy );
            myDpy = XtOpenDisplay( myContext, 0, 0, “CoEd”, 0, 0,
                           pArgc, globalArgv );
        }
        *parent = XtAppCreateShell( 0, “CoEd”,
                applicationShellWidgetClass, myDpy, 0, 0 );
            XtVaSetValues( *parent,
                   XtNmappedWhenManaged, False,
                   XtNheight, 1,
                   XtNwidth, 1,
                   0 );
```

```
        XtRealizeWidget( *parent );
}
shell = XtCreatePopupShell( "CoEd",
            applicationShellWidgetClass, *parent, 0, 0 );
XtVaSetValues( shell, XtNuserData, this, 0 );
// Pop up next to our parent
if ((_x != INT_MAX) && (_y != INT_MAX) && (_w != INT_MAX)) {
        // XXX Be smarter about picking a geometry
        Dimension x= _x + _w;
        Dimension y= _y;
        XtVaSetValues( shell, XtNx, x, XtNy, y, 0 );
}
XtAddCallback( shell, XtNdestroyCallback, CoEditor::_destroyCB_,
           this );
OlAddCallback( shell, XtNwmProtocol, CoEditor::_wmProtocolCB_, this );
_baseFrame = XtVaCreateManagedWidget(
            "baseFrame", rubberTileWidgetClass, shell, 0 );
_controls = XtVaCreateManagedWidget( "controls",
            controlAreaWidgetClass, _baseFrame,
            XtNweight, (XtArgVal)0,
            0 );
_fileBut = XtVaCreateManagedWidget( "File",
            menuButtonWidgetClass, _controls, 0 );
Widget menuPane;
XtVaGetValues( _fileBut, XtNmenuPane, &menuPane, 0 );
for (int i = 0; i < numFileButs; i++) {
        Widget but = XtVaCreateManagedWidget( fileButs[i],
            oblongButtonWidgetClass, menuPane,
            XtNuserData, i, 0 );
        XtAddCallback( but, XtNselect, CoEditor::_fileButsCB_, this );
}
_editBut = XtVaCreateManagedWidget( "Edit",
            menuButtonWidgetClass, _controls, 0 );
XtVaGetValues( _editBut, XtNmenuPane, &menuPane, 0 );
for (i = 0; i < numEditButs; i++) {
        Widget but = XtVaCreateManagedWidget( editButs[i],
            oblongButtonWidgetClass, menuPane,
            XtNuserData, i, 0 );
        XtAddCallback( but, XtNselect, CoEditor::_editButsCB_, this );
}

_scrolledWin = XtVaCreateManagedWidget(
            "scrolledWin", scrolledWindowWidgetClass,
            _baseFrame,
            XtNforceVerticalSB,(XtArgVal)True,
            0 );
```

```
    _text = (TextEditWidget)XtVaCreateManagedWidget(
                "text", textEditWidgetClass, _scrolledWin,
                0 );
    XtVaSetValues( (Widget)_text, XtNuserData, this, 0 );

    XtRealizeWidget( shell );
    XtPopup( shell, XtGrabNone );
    if (numEditors < MaxEditors) {
            editors[ numEditors ] = this;
            numEditors++;
    }
    if (numEditors >= maxBuffers) {
            tt_ptype_undeclare( "DT_CoEd" );
    }
}

Tt_status
CoEditor::_unload()
{
    Tt_status status = TT_OK;
    if (_filePats != 0) {
            // Unregister interest in ToolTalk events and destroy patterns
            status = ttdt_file_quit( _filePats, 1 );
            _filePats = 0;
    }
    if (_file != 0) {
            free( _file );
            _file = 0;
    }
    return status;
}

Tt_status
CoEditor::_load(
    const char *file
)
{
    int reloading = 1;
    if (file != 0) {
            if ((_file != 0) && (strcmp( file, _file ) != 0)) {
            reloading = 0;
            _unload();
            } else {
            _file = strdup( file );
            }
    }
```

```
    // Join a file Can be called recursively, below
if (_filePats == 0) {
        _filePats = ttdt_file_join( _file, TT_SCOPE_NONE, 1,
                    CoEditor::_fileCB_, this );
}
XtVaSetValues( (Widget)_text,
        XtNsourceType,                (XtArgVal)OL_DISK_SOURCE,
        XtNsource,                    (XtArgVal)_file,
        NULL );
_textBuf  = OlTextEditTextBuffer( _text );
RegisterTextBufferUpdate( _textBuf, CoEditor::_textUpdateCB_,
              (caddr_t)this );
if (_modifiedByMe && reloading) {
        ttdt_file_event( _contract, TTDT_REVERTED, _filePats, 1 );
}
_modifiedByMe = 0;
    // Does the file have any changes pending?
_modifiedByOther = ttdt_Get_Modified( _contract, _file, TT_BOTH,
                      10 * timeOutFactor );
if (_modifiedByOther) {
        int choice = userChoice( myContext, _baseFrame,
            "Another tool has modifications pending for "
            "this file.\nDo you want to ask it to save "
            "or revert the file?", 3, "Save", "Revert",
            "Ignore" );
        Tt_status status = TT_OK;
    switch (choice) {
        case 0:
        // Save pending changes
        status = ttdt_Save( _contract, _file, TT_BOTH,
                    10 * timeOutFactor );
        break;
        case 1:
        // Revert file to last version
        status = ttdt_Revert( _contract, _file, TT_BOTH,
                      10 * timeOutFactor );
        break;
        }
        if (status != TT_OK) {
        char *s = tt_status_message( status );
        userChoice( myContext, _baseFrame, s, 1, "Okay" );
        tt_free( s );
        } else if (choice == 0) {
        // file was saved, so reload
        return _load( 0 );
        } else if (choice == 1) {
```

```
            // file was reverted
            _modifiedByOther = 0;
            }
    }
    return TT_OK;
}

Tt_status
CoEditor::_load(
    unsigned char  *contents,
    int         //len
)
{
    _unload();
    XtVaSetValues( (Widget)_text,
            XtNsourceType,                (XtArgVal)OL_DISK_SOURCE,
            XtNsource,                    (XtArgVal)contents,
            NULL );
    _textBuf  = OlTextEditTextBuffer( _text );
    RegisterTextBufferUpdate( _textBuf, CoEditor::_textUpdateCB_,
                  (caddr_t)this );
    _modifiedByMe = 0;
    _modifiedByOther = 0;
    return TT_OK;
}

//
// Caller responsible for reporting any errors to user
//
Tt_status
CoEditor::_save()
{
    Tt_status status;
    if (_file != 0) {
            if (SaveTextBuffer( _textBuf, _file ) != SAVE_SUCCESS) {
                return TT_DESKTOP_EIO;
            }
            _modifiedByMe = 0;
            _modifiedByOther = 0;
            // File has been saved
            ttdt_file_event( _contract, TTDT_SAVED, _filePats, 1 );
    }
    if (_contract != 0) {
            int   len= 0;
            char *contents= 0;
            if (_file == 0) {
```

```
            // If you worry that the buffer might be big,
            // you could instead try a a temp file to
            // transfer the data “out of band”.
            contents = _contents( &len );
            }
            status = ttmedia_Deposit( _contract, 0, “ISO_Latin_1”,
                      (unsigned char *)contents,
                      len, _file, 10 * timeOutFactor );
            if (status != TT_OK) {
            return status;
            }
            _modifiedByMe = 0;
            _modifiedByOther = 0;
        if (contents != 0) {
                XtFree( contents );
            }
    }
    return status;
}

Tt_status
CoEditor::_revert() // XXX how about we always just send Revert? :-)
{
    if (! _modifiedByMe) {
            return TT_OK;
    }
    return _load( 0 ); // XXX what if it’s not a file? keep last deposit
}

void
CoEditor::_destroyCB_(
    Widget     w,
    XtPointer coEditor,
    XtPointer call_data
)
{
    ((CoEditor *)coEditor)->_destroyCB( w, call_data );
}

void
CoEditor::_destroyCB(
    Widget     ,
    XtPointer //call_data
)
{
    delete this;
```

```
}

void
CoEditor::_wmProtocolCB_(
    Widget    w,
    XtPointer coEditor,
    XtPointer wmMsg
)
{
    ((CoEditor *)coEditor)->_wmProtocolCB( w, (OlWMProtocolVerify*)wmMsg );
}

void
CoEditor::_wmProtocolCB(
    Widget    w,
    OlWMProtocolVerify     *wmMsg
)
{
    switch (wmMsg->msgtype) {
        case OL_WM_DELETE_WINDOW:
            if (_modifiedByMe) {
            int choice =
                userChoice( myContext, _baseFrame,
                        "The text has unsaved changes.",
                        3, "Save, then Quit",
                        "Discard, then Quit",
                        "Cancel" );
            switch (choice) {
                case 0:
                break;
                case 1:
                _revert();
                break;
                case 2:
                return;
            }
            }
            if (numEditors > 1) {
            XtDestroyWidget( shell );
            } else {
            // XXX OlWmProtocolAction() doesn't call destructor?!
            delete this;
            OlWMProtocolAction( w, wmMsg, OL_DEFAULTACTION );
            }
            break;
        default:
```

```
            OlWMProtocolAction( w, wmMsg, OL_DEFAULTACTION );
            break;
    }
}

void
CoEditor::_fileButsCB_(
    Widget    button,
    XtPointer coEditor,
    XtPointer call_data
)
{
    ((CoEditor *)coEditor)->_fileButsCB( button, call_data );
}

void
CoEditor::_fileButsCB(
    Widget    button,
    XtPointer //call_data
)
{
    FileOp op;
    XtVaGetValues( button, XtNuserData, &op, 0 );
    Tt_status status = TT_OK;
    switch (op) {
        case Open:
            break;
        case Revert:
            status =_revert();
            break;
        case Save:
            status =_save();
            break;
        case SaveAs:
            break;
    }
    if (status != TT_OK) {
            _adviseUser( status );
    }
}

void
CoEditor::_editButsCB_(
    Widget    button,
    XtPointer coEditor,
    XtPointer call_data
```

```
)
{
    ((CoEditor *)coEditor)->_editButsCB( button, call_data );
}

void
CoEditor::_editButsCB(
    Widget    button,
    XtPointer //call_data
)
{
    EditOp op;
    XtVaGetValues( button, XtNuserData, &op, 0 );
    Tt_status status = TT_OK;
    switch (op) {
            int     len;
            char       *contents;
            const char     *mediaType;
            Tt_messagemsg;
            Tt_pattern     *pats;
        case SelText:
        case SelAppt:
            if (op == SelText) {
                mediaType = "ISO_Latin_1";
            } else {
            mediaType = "DT_CM_Appointment";
            }
            //contents = _selection( &len );
            contents = _contents( &len );
            if (len <= 0) {
            return;
            }
            // Media load callback
            msg = ttmedia_load( _contract, CoEditor::_mediaLoadMsgCB_,
                    this, TTME_EDIT, mediaType,
                    (unsigned char *)contents, len, 0, 0, 1 );
            if (contents != 0) {
            XtFree( contents );
            }
            status = tt_ptr_error( msg );
            if (status != TT_OK) {
            break;
            }
            pats = ttdt_subcontract_manage( msg, 0, shell, this );
            status = tt_ptr_error( pats );
            if (status != TT_OK) {
```

```
            break;
            }
            break;
    }
    if (status != TT_OK) {
            char *s = tt_status_message( status );
            char buf[ 1024 ];
            sprintf( buf, "%d: %s", status, s );
            tt_free( s );
            userChoice( myContext, _baseFrame, buf, 1, "Okay" );
    }
}

char *
CoEditor::_contents(
    int *len
)
{
    _textBuf  = OlTextEditTextBuffer( _text );
    TextLocation  start    = { 0, 0, 0 };
    TextLocation  end      = LastTextBufferLocation( _textBuf );
    char     *contents = GetTextBufferBlock( _textBuf, start, end );

    *len = 0;
    if (contents != 0) {
            *len = strlen( contents );
    }
    return contents;
}

Tt_status
CoEditor::_acceptContract(
    Tt_message msg
)
{
    static const char *here = "CoEditor::_acceptContract()";

    _contract = msg;
    if (tt_message_status( msg ) == TT_WRN_START_MESSAGE) {
            //
            // Join session before accepting start message,
            // to prevent unnecessary starts of our ptype
            //
            Widget session_shell = shell;
            if (maxBuffers > 1) {
            //
```

```
            // If we are in multi-window mode, just use
            // our unmapped toplevel shell as our session
            // shell, since we do not know if any particular
            // window will exist the whole time we are in
            // the session.
            //
            session_shell = XtParent(shell );
            }
            // Join the session and register patterns and callbacks
            sessPats = ttdt_session_join( 0, 0, session_shell, this, 1 );
    }
        // Accept responsibility to handle a request
    _contractPats = ttdt_message_accept(
                msg, CoEditor::_contractCB_, shell, this,
                1, 1 );
    Tt_status status = tt_ptr_error( _contractPats );
    if (status != TT_OK) {
            return status;
    }
    return status;
}

Tt_message
CoEditor::_contractCB_(
    Tt_message,                             //msg,
    Tttk_op ,                               //op,
    Widget  ,                               //shell,
    void        *,                          //coEditor,
    Tt_message                              //Contract
)
{
    return 0;
}

void
CoEditor::_editButCB_(
    Widget    w,
    XtPointer coEditor,
    XtPointer call_data
)
{
    ((CoEditor *)coEditor)->_editButCB( w, call_data );
}


void
```

```
CoEditor::_editButCB(
    Widget    ,
    XtPointer //call_data
)
{
    int        len;
    char   *contents = _contents( &len );
        // Media Load Callback
    Tt_message msg = ttmedia_load( _contract, CoEditor::_mediaLoadMsgCB_,
                       this, TTME_EDIT, "ISO_Latin_1",
                       (unsigned char *)contents,
                       len, 0, 0, 1 );
    if (contents != 0) {
            XtFree( contents );
    }
    Tt_pattern *pats = ttdt_subcontract_manage( msg, 0, shell, this );
}


Tt_message
CoEditor::_mediaLoadMsgCB_(
    Tt_message                msg,
    Tttk_op                   op,
    unsigned char  *contents,
    int             len,
    char           *file,
    void           *clientData
)
{
    return ((CoEditor *)clientData)->_mediaLoadMsgCB( msg, op,
                       contents, len, file );
}

Tt_message
CoEditor::_mediaLoadMsgCB(
    Tt_message                msg,
    Tttk_op,
    unsigned char  *contents,
    int                       len,
    char           *file
)
{
    if (len > 0) {
            XtVaSetValues( (Widget)_text,
            XtNsourceType,                (XtArgVal)OL_STRING_SOURCE,
            XtNsource,             (XtArgVal)contents,
```

```
            NULL );
            _textBuf  = OlTextEditTextBuffer( _text );
            RegisterTextBufferUpdate( _textBuf, CoEditor::_textUpdateCB_,
                    (caddr_t)this );
            // ReplaceBlockInTextBuffer
    } else if (file != 0) {
    }
    tt_message_destroy( msg );
    return 0;
}

void
CoEditor::_textUpdateCB_(
    XtPointer         coEditor,
    XtPointer         pTextBuffer,
    EditResult        status
)
{
    if (coEditor == 0) {
            return;
    }
    ((CoEditor *)coEditor)->_textUpdateCB( (TextBuffer *)pTextBuffer,
                          status );
}

void
CoEditor::_textUpdateCB(
    TextBuffer     *textBuf,
    EditResult                //editStatus
)
{
    //Tt_status status;
    if (_textBuf != textBuf) {
            fprintf( stderr, "_textBuf != textBuf" );
    }
    if ((! _modifiedByMe) && TextBufferModified( _textBuf )) {
            _modifiedByMe = TRUE;
            // File has changes pending
            ttdt_file_event( _contract, TTDT_MODIFIED, _filePats, 1 );
    }
}

Tt_message
CoEditor::_fileCB_(
    Tt_message                msg,
    Tttk_op                   op,
```

```
    char            *pathname,
    void        *coEditor,
    int                 trust,
    int                 me
)
{
    tt_free( pathname );
    if (coEditor == 0) {
            return msg;
    }
    return ((CoEditor *)coEditor)->_fileCB( msg, op, pathname,
                        trust, me );
}

Tt_message
CoEditor::_fileCB(
    Tt_message                msg,
    Tttk_op                   op,
    char        *pathname,
    int,                          //trust
    int                           //me
)
{
    tt_free( pathname );
    Tt_status status = TT_OK;
    switch (op) {
        case TTDT_MODIFIED:
            if (_modifiedByMe) {
            // Hmm, the other editor either doesn't know or
            // doesn't care that we are already modifying the
            // file, so the last saver will win.
            // XXX Or: a race condition has arisen!
            } else {
            // Interrogate user if she ever modifies the buffer
            _modifiedByOther = 1;
            XtAddCallback( (Widget)_text, XtNmodifyVerification,
                (XtCallbackProc)CoEditor::_textModifyCB_, 0 );
            }
            break;
        case TTDT_GET_MODIFIED:
            tt_message_arg_ival_set( msg, 1, _modifiedByMe );
            tt_message_reply( msg );
            break;
        case TTDT_SAVE:
            status = _save();
            if (status == TT_OK) {
```

```
            tt_message_reply( msg );
            } else {
            // Fail message
            tttk_message_fail( msg, status, 0, 0 );
            }
            break;
        case TTDT_REVERT:
            status = _revert();
            if (status == TT_OK) {
            tt_message_reply( msg );
            } else {
            // Fail message
            tttk_message_fail( msg, status, 0, 0 );
            }
            break;
        case TTDT_REVERTED:
        case TTDT_SAVED:
        case TTDT_MOVED:
        case TTDT_DELETED:
            printf( "CoEditor::_fileCB(): %s\n", tttk_op_string( op ));
            break;
    }
    tt_message_destroy( msg );
    return 0;
}

void
CoEditor::_textModifyCB_(
    TextEditWidget              text,
    XtPointer                   ,
    OlTextModifyCallData *mod
)
{
    CoEditor *coEditor = 0;
    XtVaGetValues( (Widget)text, XtNuserData, &coEditor, 0 );
    if (coEditor == 0) {
            return;
    }
    coEditor->_textModifyCB( mod );
}

void
CoEditor::_textModifyCB(
    OlTextModifyCallData *mod
)
{
```

```
    if (_modifiedByOther != 1) {
            return;
    }
    int cancel = userChoice( myContext, _baseFrame,
               "Another tool has modifications pending for this file.\n"
               "Are you sure you want to start modifying the file?",
               2, "Modify", "Cancel" );
    if (cancel) {
            mod->ok = FALSE;
    }
    _modifiedByOther = 2;
}

void
CoEditor::_adviseUser(
    Tt_status status
)
{
    char *s = tt_status_message( status );
    char buf[ 1024 ];
    sprintf( buf, "%d: %s", status, s );
    tt_free( s );
    userChoice( myContext, _baseFrame, buf, 1, "Okay" );
}
```

New ToolTalk Functions

C

This chapter describes ToolTalk functions that are new for this release. To use these functions, you need to include the ToolTalk header file:

```
#include <Tt/tt_c.h>
```

tt_error

```
void tt_error(const char *funcname, Tt_status status)
```

The `tt_error` function is a publicly-known null function. This function is called by the ToolTalk library just before it returns from any ToolTalk API call that has a status other than `TT_OK`. The name of the function that is about to return and the status code is passed. You can use this call to set a `dbx` breakpoint in `tt_error` to quickly catch and trace back any ToolTalk errors. You can also interpose this function, for example, to log ToolTalk errors to `stderr`. The following code example shows how an application might do this.

```
void tt_error(const char *funcname, Tt_status status) {
    fprintf(stderr, "ToolTalk function %s returned %s.\n",
            funcname, tt_status_message(status));
}
```

tt_file_netfile

```
char *        tt_file_netfile(     const char *   filename );
```

The tt_file_netfile function maps between local and canonical path names. It converts the file specified in filename to a netfilename that can be passed to other hosts on the network. The filename is an absolute or relative path name that is valid on the local host. The last component of filename is not required; however, every other component of filename must exist.

Note – You do not need to call the tt_open function before you use this function.

This function returns either an error pointer or, if successful, a newly-allocated null-terminated string of an unspecified format, which may be passed to the tt_netfile_file function.

Use tt_ptr_error to extract a status from an error pointer. Possible errors are described in Table C-1.

Table C-1 Possible Errors Returned by tt_file_netfile

Error	Description
TT_ERR_PATH	filename is a path that is not valid on this host.
TT_ERR_DBAVAIL	rpc.ttdbserverd could not be reached on host.
TT_ERR_DBEXIST	rpc.ttdbserverd does not appear to be properly installed on host.

To free allocated strings, use either the tt_free or tt_release call.

To convert the file back to a local file name for the same file, use the tt_netfile_file function.

tt_host_file_netfile

```
char *tt_host_file_netfile(  const char *  host,
                             const char *  filename );
```

The tt_host_file_netfile function maps between local and canonical path names on a remote host. It converts the file specified in host to a netfilename that can be passed to other hosts on the network. The filename is an absolute or relative path name that is valid on the remote host. The last component of filename is not required; however, every other component of filename must exist.

Note – You do not need to call the tt_open function before you use this function.

This function returns either an error pointer or, if successful, a newly-allocated null-terminated string of an unspecified format, which may be passed to the tt_netfile_file function.

Use tt_ptr_error to extract a status from an error pointer. Possible errors are described in Table C-2.

Table C-2 Possible Errors Returned by tt_host_file_netfile

Error	Description
TT_ERR_PATH	filename is a path that is not valid on the remote host.
TT_ERR_DBAVAIL	rpc.ttdbserverd could not be reached on host.
TT_ERR_DBEXIST	rpc.ttdbserverd does not appear to be properly installed on host.
TT_ERR_UNIMP	rpc.ttdbserverd version does not support the tt_host_file_netfile function.

- To free allocated strings, use either the tt_free or tt_release call.

To convert the file back to a local file name for the same file, use the tt_host_netfile_file function.

tt_host_netfile_file

```
char *tt_host_netfile_file( const char *  host,
                            const char *  netfilename );
```

The `tt_host_file_netfile` function maps between local and canonical path names on the remote host. It converts the file specified `netfilename` to a path name that is valid on the remote host. The `netfilename` is a copy of a null-terminated string returned by the `tt_netfile_file` function.

Note – You do not need to call the `tt_open` function before you use this function.

If the specified file is not currently mounted on the local host, a path name in the form of

/DTMOUNTPOINT/host/filepath

is constructed, where:

`DTMOUNTPOINT` is the intended mount point for the automounter's host map. You can also specify this mount point with the environment variable `DTMOUNTPOINT`.

`host` is the host that contains the file.

`filepath` is the path to the file contained on the host.

This function returns either an error pointer or, if successful, a newly-allocated null-terminated local file name.

Use `tt_ptr_error` to extract a status from an error pointer. Possible errors are described in Table C-3.

Table C-3 Possible Errors Returned by tt_host_netfile_file

Errors	Description
TT_ERR_PATH	netfilename is not a valid netfilename.
TT_ERR_DBAVAIL	rpc.ttdbserverd could not be reached on host.
TT_ERR_DBEXIST	rpc.ttdbserverd does not appear to be properly installed on host.
TT_ERR_UNIMP	rpc.ttdbserverd version does not support the tt_host_netfile_file function.

To free allocated strings, use either the `tt_free` or `tt_release` call.

To convert the file back to a local file name for the same file, use the `tt_host_file_netfile` function.

tt_message_print

```
char *  tt_message_print(Tt_message m);
```

The `tt_message_print` function allows you to print out messages that are received by not understood.

To free allocated strings, use either the `tt_free` or `tt_release` call.

This function returns either the error `TT_ERR_POINTER` or, if successful, the message `m` in a buffer allocated by ToolTalk (in the same manner as is done in other ToolTalk API calls such as `tt_X_session`).

tt_netfile_file

```
char *        tt_netfile_file(     const char *   netfilename );
```

The tt_netfile_file function maps between canonical and local path names. It converts the file specified netfilename to a path name that is valid on the local host. The netfilename is a copy of a null-terminated string returned by tt_netfile_file.

Note – You do not need to call the tt_open function before you use this function.

If the specified file is not currently mounted on the local host, a path name in the form of

/DTMOUNTPOINT/host/filepath

is constructed, where:

DTMOUNTPOINT is the intended mount point for the automounter's host map. You can also specify this mount point with the environment variable DTMOUNTPOINT.

host is the host that contains the file.

filepath is the path to the file contained on the host.

This function returns either an error pointer or, if successful, a newly-allocated null-terminated local file name.

Use tt_ptr_error to extract a status from an error pointer. Possible errors are described in Table C-4.

Table C-4 Possible Errors Returned by tt_netfile_file

Error	Description
TT_ERR_PATH	netfilename is not a valid netfilename.
TT_ERR_DBAVAIL	rpc.ttdbserverd could not be reached on host.
TT_ERR_DBEXIST	rpc.ttdbserverd does not appear to be properly installed on host.

To free allocated strings, use either the tt_free or tt_release call.

To convert the file back to a net file name for the same file, use the `tt_file_netfile` function.

tt_pattern_print

```
char *  tt_message_print(Tt_pattern p);
```

The tt_pattern_print function allows you to print out patterns.

To free allocated strings, use either the `tt_free` or `tt_release` call.

This function returns either the error `TT_ERR_POINTER` or, if successful, the pattern `p` in a buffer allocated by ToolTalk (in the same manner as is done in other ToolTalk API calls such as `tt_X_session`).

Examples D

Example Ttdt_contract_cb

The code example is a typical algorithm of a `Ttdt_contract_cb` callback for an application that handles its own Pause/Resume/Quit requests but allows the toolkit to handle the X11-related requests.

Note – This example callback deals with the case when the *contract* parameter has a value other than zero and can, therefore, also be used as the `Ttdt_contract_cb` callback passed to `ttdt_message_accept`.

```
Tt_message
myContractCB(
        Tt_message      msg,
        void           *clientdata,
        Tt_message      contract
)
{
        char *opString = tt_message_op( msg );
        Tttk_op op = tttk_string_op( opString );
        tt_free( opString );
        int silent = 0;
        int force  = 0;
        Boolean cancel = False;
        Boolean sensitive = True;
        char *status, command;
        switch (op) {
            case TTDT_QUIT:
```

```
    tt_message_arg_ival( msg, 0, &silent );
    tt_message_arg_ival( msg, 1, &force );
    if (contract == 0) {
            /* Quit entire application */
            cancel = ! myQuitWholeApp( silent, force );
    } else {
            /* Quit just the specified request being worked on */
            cancel = ! myCancelThisRequest(contract, silent, force);
    }
    if (cancel) {
            /* User canceled Quit; fail the Quit request */
            tttk_message_fail( msg, TT_DESKTOP_ECANCELED, 0, 1 );
    } else {
            tt_message_reply( msg );
            tttk_message_destroy( msg );
    }
    return 0;
case TTDT_PAUSE:
    sensitive = False;
case TTDT_RESUME:
    if (contract == 0) {
            int already = 1;
            if (XtIsSensitive( myTopShell ) != sensitive) {
                    already = 0;
                    XtSetSensitive( myTopShell, sensitive );
            }
            if (already) {
                    tt_message_status_set(msg,TT_DESKTOP_EALREADY);
            }
    } else {
            if (XtIsSensitive( thisShell ) == sensitive) {
                    tt_message_status_set(msg,TT_DESKTOP_EALREADY);
            } else {
                    XtSetSensitive( thisShell, sensitive );
            }
    }
    tt_message_reply( msg );
    tttk_message_destroy( msg );
    return 0;
case TTDT_GET_STATUS:
    if (contract == 0) {
            status = "Message about status of entire app";
    } else {
            status = "Message about status of this request";
    }
    tt_message_arg_val_set( msg, 0, status );
```

```
                tt_message_reply( msg );
                tttk_message_destroy( msg );
                return 0;
            case TTDT_DO_COMMAND:
                if (! haveExtensionLanguage) {
                        tttk_message_fail( msg, TT_DESKTOP_ENOTSUP, 0, 1 );
                        return 0;
                }
                command = tt_message_arg_val( msg, 0 );
                result = myEval( command );
                tt_free( command );
                tt_message_status_set( msg, result );
                if (tt_is_err( result )) {
                        tttk_message_fail( msg, result, 0, 1 );
                } else {
                        tt_message_reply( msg );
                        tttk_message_destroy( msg );
                }
                return 0;
        }
        /* Unrecognized message; do not consume it */
        return msg;
}
```

Example Ttdt_file_cb

The code example below is a typical algorithm of this callback.

```
Tt_message
myFileCB(
        Tt_message      msg,
        Tttk_op         op,
        char           *pathname,
        int             trust,
        int             isMe
)
{
        tt_free( pathname );
        Tt_status status = TT_OK;
        switch (op) {
```

```
case TTDT_MODIFIED:
    if ((_modifiedByMe) && (! isMe)) {
            // Hmm, the other editor either does not know or
            // does not care that we are already modifying the
            // file, so the last saver will win.
    } else {
            // Interrogate user if she ever modifies the buffer
            _modifiedByOther = 1;
            XtAddCallback( myTextWidget, XmNmodifyVerifyCallback,
                           myTextModifyCB, 0 );
    }
    break;
case TTDT_GET_MODIFIED:
    tt_message_arg_ival_set( msg, 1, _modifiedByMe );
    tt_message_reply( msg );
    break;
case TTDT_SAVE:
    status = mySave( trust );
    if (status == TT_OK) {
            tt_message_reply( msg );
    } else {
            tttk_message_fail( msg, status, 0, 0 );
    }
    break;
case TTDT_REVERT:
    status = myRevert( trust );
    if (status == TT_OK) {
            tt_message_reply( msg );
    } else {
            tttk_message_fail( msg, status, 0, 0 );
    }
    break;
case TTDT_REVERTED:
    if (! isMe) {
            _modifiedByOther = 0;
    }
    break;
case TTDT_SAVED:
    if (! isMe) {
            _modifiedByOther = 0;
            int choice = myUserChoice( myContext, myBaseFrame,
                                 "Another tool has saved "
                                 "this file.", 2, "Ignore",
                                 "Revert" );
            switch (choice) {
                case 1:
```

```
                        myRevert( 1 );
                        break;
                    }
                }
                break;
            case TTDT_MOVED:
            case TTDT_DELETED:
                // Do something appropriate
                break;
        }
        tttk_message_destroy( msg );
        return 0;
}
```

Example Ttmedia_load_msg_cb

The code example below is a typical algorithm of this callback.

```
Tt_message
myLoadMsgCB(
    Tt_messagemsg,
    void            *clientData,
    Tttk_op op,
    unsigned char   *contents,
    int     len,
    char            *file
)
{
    if (len > 0) {
        // Replace data with len bytes in contents
    } else if (file != 0) {
        // Replace data with data read from file
    }
    if (op == TTME_DEPOSIT) {
        tt_message_reply( msg );
    }
    tttk_message_destroy( msg );
    return 0;
}
```

Example Ttmedia_load_pat_cb

The code example below is a typical algorithm of this callback.

```
Tt_message
myAcmeSheetLoadCB(
        Tt_message      msg,
        void           *client_data,
        Tttk_op         op,
        Tt_status       diagnosis,
        unsigned char  *contents,
        int             len,
        char           *file,
        char           *docname
)
{
        Tt_status status = TT_OK;
        if (diagnosis != TT_OK) {
                // toolkit detected an error
                if (tt_message_status( msg ) == TT_WRN_START_MESSAGE) {
                        //
                        // Error is in start message!  We now have no
                        // reason to live, so tell main() to exit().
                        //
                        myAbortCode = 2;
                }
                // let toolkit handle the error
                return msg;
        }
        if ((op == TTME_COMPOSE) && (file == 0)) {
                // open empty new buffer
        } else if (len > 0) {
                // load contents into new buffer
        } else if (file != 0) {
                if (ttdt_Get_Modified( msg, file, TT_BOTH, myCntxt, 5000 )) {
                        switch (myUserChoice( "Save, Revert, Ignore?" )) {
                            case 0:
                                ttdt_Save( msg, file, TT_BOTH, myCntxt, 5000 );
                                break;
                            case 1:
                                ttdt_Revert( msg, file, TT_BOTH, myCntxt, 5000);
                                break;
                        }
                }
                // load file into new buffer
```

```
        } else {
                tttk_message_fail( msg, TT_DESKTOP_ENODATA, 0, 1 );
                tt_free( contents ); tt_free( file ); tt_free( docname );
                return 0;
        }
        int w, h, x, y = INT_MAX;
        ttdt_sender_imprint_on( 0, msg, 0, &w, &h, &x, &y, myCntxt, 5000 );
        positionMyWindowRelativeTo( w, h, x, y );
        if (maxBuffersAreNowOpen) {
                // Un-volunteer to handle future requests until less busy
                tt_ptype_undeclare( "Acme_Calc" );
        }
        if (tt_message_status( msg ) == TT_WRN_START_MESSAGE) {
                //
                // Join session before accepting start message,
                // to prevent unnecessary starts of our ptype
                //
                ttdt_session_join( 0, myContractCB, myShell, 0, 1 );
        }
        ttdt_message_accept( msg, myContractCB, myShell, 0, 1, 1 );
        tt_free( contents ); tt_free( file ); tt_free( docname );
        return 0;
}
```

Example Ptype Signature for Ttmedia_ptype_declare Function

The code example below is the signature layout of a media ptype.

```
ptype Acme_Calc {
    start "acalc";
    handle:
        /*
         * Display Acme_Sheet
         * Include in tool's ptype if tool can display a document.
         */
        session Display( in    Acme_Sheet  contents     ) => start opnum = 1;
        session Display( in    Acme_Sheet  contents,
                         in    messageID   counterfoil  ) => start opnum = 2;
        session Display( in    Acme_Sheet  contents,
                         in    title       docName      ) => start opnum = 3;
        session Display( in    Acme_Sheet  contents,
```

```
                  in    messageID   counterfoil,
                  in    title       docName      ) => start opnum = 4;
        /*
         * Edit Acme_Sheet
         * Include in tool’s ptype if tool can edit a document.
         */
        session Edit(    inout Acme_Sheet  contents     ) => start opnum = 101;
        session Edit(    inout Acme_Sheet  contents,
                         in    messageID   counterfoil  ) => start opnum = 102;
        session Edit(    inout Acme_Sheet  contents,
                         in    title       docName      ) => start opnum = 103;
        session Edit(    inout Acme_Sheet  contents,
                         in    messageID   counterfoil,
                         in    title       docName      ) => start opnum = 104;
        /*
         * Compose Acme_Sheet
         * Include in tool’s ptype if tool can compose a document from scratch.
         */
        session Edit(    out   Acme_Sheet  contents     ) => start opnum = 201;
        session Edit(    out   Acme_Sheet  contents,
                         in    messageID   counterfoil  ) => start opnum = 202;
        session Edit(    out   Acme_Sheet  contents,
                         in    title       docName      ) => start opnum = 203;
        session Edit(    out   Acme_Sheet  contents,
                         in    messageID   counterfoil,
                         in    title       docName      ) => start opnum = 204;
        /*
         * Mail Acme_Sheet
         * Include in tool’s ptype if tool can mail a document.
         */
        session Mail(    in    Acme_Sheet  contents     ) => start opnum = 301;
        session Mail(    inout Acme_Sheet  contents     ) => start opnum = 311;
        session Mail(    inout Acme_Sheet  contents,
                         in    title       docName      ) => start opnum = 313;
        session Mail(    out   Acme_Sheet  contents     ) => start opnum = 321;
        session Mail(    out   Acme_Sheet  contents,
                         in    messageID   counterfoil  ) => start opnum = 323;
};
```

Example for Xt Input Handler Function

The code example below is the Xt input handler function.

```
int myTtFd;
char *myProcID;
myProcID = ttdt_open( &myTtFd, "WhizzyCalc", "Acme", "1.0", 1 );
/* ... */
/* Process the message that started us, if any */
tttk_Xt_input_handler( myProcID, 0, 0 );
/* ... */
XtAppAddInput( myContext, myTtFd, (XtPointer)XtInputReadMask,
               tttk_Xt_input_handler, myProcID );
```

Index

F

G

H

I

L

Q

R

S

T