TriTeal Enterprise Desktop (TED™) 4.0

Tooltalk Programmer's Guide

Printed in USA

Part No. TED015004000

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/

Please
Recycle

Contents

Preface

This manual describes the application programming interface (API) components, commands, and error messages of the ToolTalk® service and how you modify your application to send and receive ToolTalk messages.

The ToolTalk service supports several messaging styles. A sender can address a ToolTalk message to a particular process, to any interested process, to an object, or to an object type. Message senders are not concerned with the locations of processes and objects in any network; the ToolTalk service finds receiving processes and objects.

Who Should Use this Book

This guide is for:

- developers who want to write applications that directly use ToolTalk messaging. (You can get many of the benefits of ToolTalk messaging by writing actions that send ToolTalk messages.)
- developers who create or maintain applications that use the ToolTalk service to inter-operate with other applications.
- system administrators who set up workstations.

This guide assumes familiarity with operating system commands, system administrator commands, and system terminology.

How This Manual Is Organized

This manual is organized into three general parts: Chapters 1 through 5 introduce ToolTalk and cover concepts and file system layout; Chapters 6 through 17 cover the ToolTalk API; and Chapters 18 through 20 cover message sets and the messaging toolkit.

This manual contains the following chapters and appendix:

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

Chapter 2, "ToolTalk Service Overview", describes applications and ToolTalk components.

Chapter 3, "Setting Up and Maintaining the ToolTalk Processes", describes ToolTalk file locations, hardware and software requirements, how to find ToolTalk version information, and installation for the ToolTalk database server.

Chapter 4, "Maintaining Application Information", describes how to maintain application information.

Chapter 5, "Maintaining Files and Objects Referenced in ToolTalk Messages", describes how to maintain file references in ToolTalk messages; how system administrators and users maintain ToolTalk objects; and how to perform maintenance on ToolTalk databases.

Chapter 6, "Participating in ToolTalk Sessions", describes the location of the ToolTalk application programming interface (API) header file; how you initialize your application and start a session with the ToolTalk service; how you provide file and session information to the ToolTalk service; how to manage storage and handle errors; and how to unregister your message patterns and close your communication with the ToolTalk service when your process is ready to quit.

Chapter 7, "Sending Messages", describes the complete ToolTalk message structure; the ToolTalk message delivery algorithm; how to create, fill in, and send a ToolTalk message; and how to attach to requests a callback that will automatically call your callback routine when the reply to your request is delivered to your application.

Chapter 8, "Message Patterns", describes message pattern attributes.

Chapter 9, “Dynamic Message Patterns", describes how to create a dynamic message pattern and register it with the ToolTalk service; and how to add callbacks to your dynamic message patterns.

Chapter 10, “Static Message Patterns", describes how to provide process and object type information at installation time; how to make a static message pattern available to the ToolTalk Service; and how to declare a ptype.

Chapter 11, “Receiving Messages", describes how to retrieve messages delivered to your application; how to handle the message once you have examined it; how to send replies; and when to destroy messages.

Chapter 12, “Objects", describes how to create ToolTalk specification objects for the objects your process creates and manages.

Chapter 13, “Managing Information Storage", describes how to manage and remove objects.

Chapter 14, “Handling Errors", describes how to handle error conditions, errors that may occur during initialization, and Tooltalk error messages found in the message catalog.

Chapter 15, “The ToolTalk Enumerated Types", describes each of the ToolTalk enumerated types.

Chapter 16, “The ToolTalk Functional Groupings", lists, in table format, the ToolTalk functions by the specific operations they perform.

Chapter 17, “ToolTalk Functions/Commands", lists, in table format, each of the ToolTalk functions and commands, along with their descriptions.

Chapter 18, “Using ToolTalk Messaging", describes how to use ToolTalk messaging.

Chapter 19, “The Messaging Toolkit", describes the Messaging Toolkit, which is a higher-level interface of the ToolTalk API.

Chapter 20, “ToolTalk Message Sets", describes the ToolTalk Desktop Services Message Set and the ToolTalk Document and Media Exchange Message Set that have been developed to help you develop applications that follow the same protocol as other applications with which your application wants to inter-operate.

Appendix A, "Frequently Asked Questions", provides the answers to some frequently asked questions about the ToolTalk service.

Related Documentation

The following is a list of related ToolTalk documentation:

- ToolTalk Messaging Overview.

Introducing the ToolTalk Service 1

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.

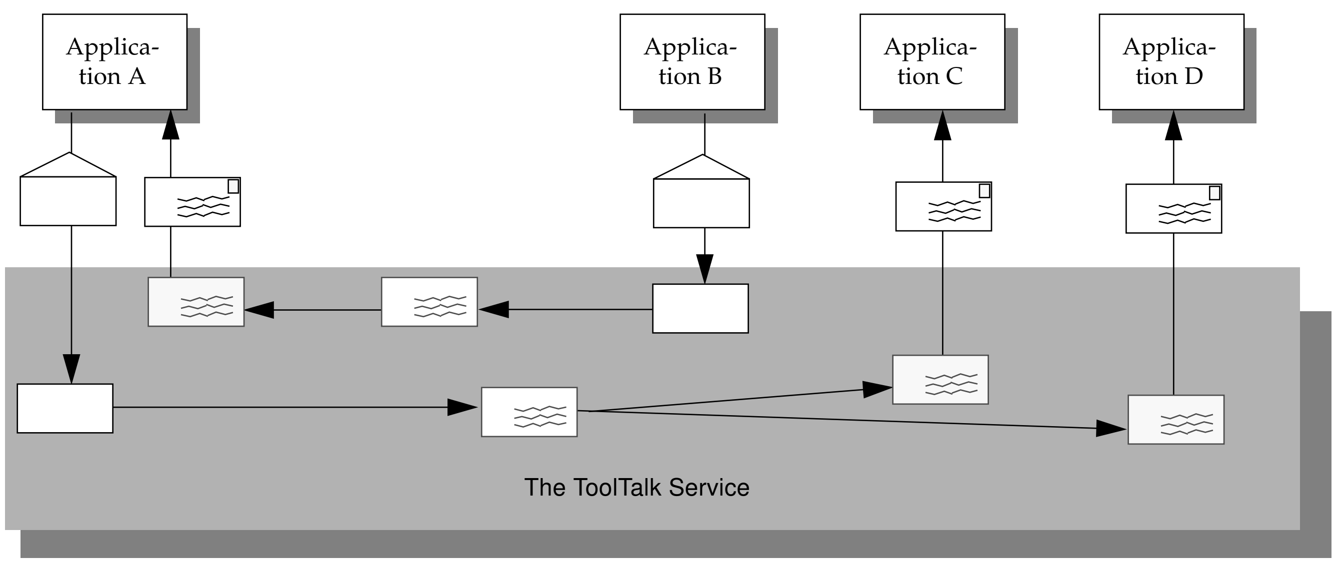

Figure 1-1 Applications Using the ToolTalk Service

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; 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.

ToolTalk Scenarios

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 illustrates two scenarios that show how the Desktop Services Message Set might be implemented.

The Smart Desktop

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. An example of this application-level program (known as *smart desktops*) is the TriTeal Enterprise Desktop (TED) 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 encompasses 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. Quinn double clicks on the File Manager icon.
    - The File Manager opens and displays the files in Quinn's current directory.

2. Quinn 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 Quinn'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 match, 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. Quinn 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 or in horizontal environments (such as compound documents). Common to both of these applications is the premise that the overall solution is built out of 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 illustrates how Desktop Services Message Set might be implemented as an integrated toolset:

1. Kaia is working on a compound document using her favorite editor.

   She decides to change some of the source code text.
2. Kaia double clicks on the source code text.
   a. The Document Editor first determines that the text represents source code and then determines what 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 Kaia'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 match, 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. Kaia 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 applications 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. Dana 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. Dana 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, Dana clicks once on the icon to select it and uses the third mouse button to bring up an Edit menu.

   An editing application is launched and Dana 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 audiotool to play a sound file as a reminder of an appointment, as illustrated in the following scenario:

1. Mollie opens her calendar manager and sets an appointment.
2. Mollie clicks on an audio response button, which causes the soundtool to pop up.
3. Mollie records her message; for example, “Bring the report.”

When Mollie’s appointment reminder is executed, the calendar manager will start the audiotool and play Mollie’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. Art opens two documents that are different media types.
2. Art selects a portion of *Document A* and cuts the portion using the standard *X*-windowing *cut* facility.
3. Art 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 B*, it requests 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 registered 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 delivers 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.

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 that 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 application programming interface (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.

ToolTalk Service Overview 2

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.

ToolTalk Architecture

The following ToolTalk service components work together to provide inter-application communication and object information management:

- `ttsessiond` is the ToolTalk communication process.

  This process joins together senders and receivers that are either using the same X server or interested in the same file. One ttsession communicates with other ttsessions when a message needs to be delivered to an application in another session.

- `rpc.ttdbserverd` is the ToolTalk database server process.

  One `rpc.ttdbserverd` is installed on each machine that contains a disk partition that stores files of interest to ToolTalk clients or files that contain ToolTalk objects.

  File and ToolTalk object information is stored in a records database managed by `rpc.ttdbserverd`.

- `libtt` is the ToolTalk application programming interface (API) library.

  Applications include the API library in their program and call the ToolTalk functions in the library.

Applications provide the ToolTalk service with process and object type information. This information is stored in an XDR format file, which is referred to as the ToolTalk Types Database in this manual.

Figure 2-1 illustrates the ToolTalk service architecture.

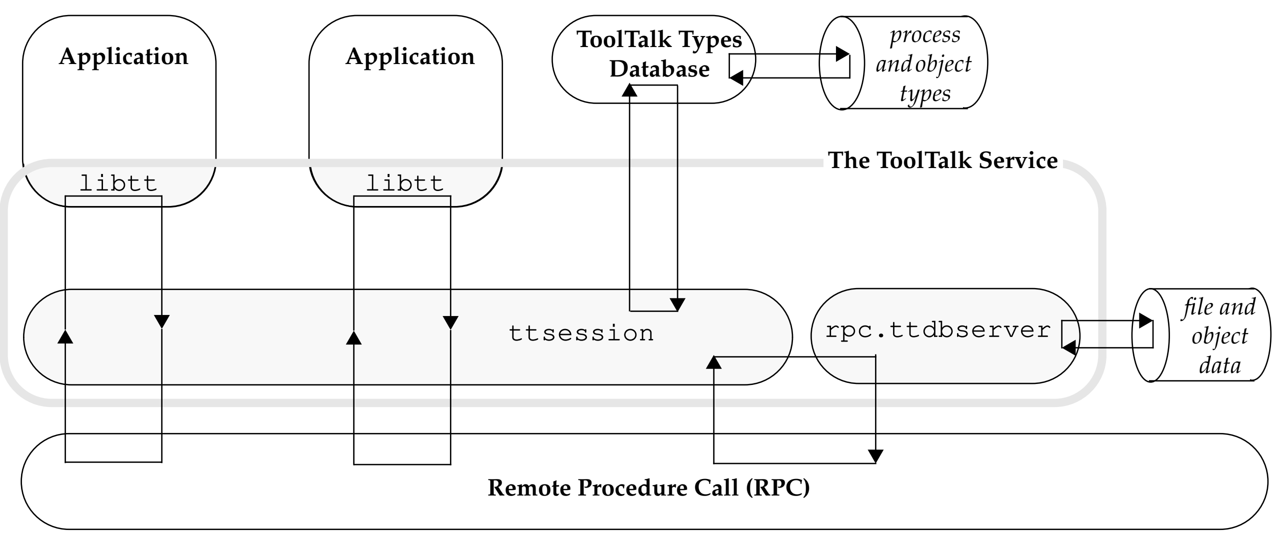

Figure 2-1 ToolTalk Service Architecture

Starting a ToolTalk Session

The ToolTalk message server, `ttsession`, automatically starts when you open communication with the ToolTalk server or when TED is started. This background process must be running before any messages can be sent or received. Each message server defines a session.

Note – A session can have more than one session identifier.

To manually start a session, enter the following command on the command line:

`ttsession [-hNpsStv][-E| -X][-a` *level*`][-d` *display*`][-c [`*command]*`]`

See Table 2-1 for a description of the `ttsession` command line options.

Table 2-1 `ttsession` Command Line Options

Argument	**Description**
`-a` *level*	Sets the server authentication level. The level must be *unix* or *des*.
`-c` *command*	Start process tree session and run the given command. The *ttsession* utility sets the environment variable `TT_SESSION` to the name of this session. Any process started with this variable in the environment, defaults to being in this session. If *command* is omitted, `ttsession` invokes the shell named by the `SHELL` environment variable. Everything after `-c` on the command line is used as the command to be executed.
`-d` *display*	Directs `ttsession` to start an `X` session for the given display. Normally, `ttsession` uses the `$DISPLAY` environment variable.
`-E`	Read in the types from the Classing Engine data base. If neither `-E` nor `-X` is given, `-X` is assumed.
`-h`	Write a help message to standard error that describes the command syntax of `ttsession,` and exit.
`-N`	Maximize the number of clients allowed to (in other words, open procids in) this session by attempting to raise the limit of open file descriptors. The precise number of clients is system-dependent; on some systems this options may have no effects.
`-p`	Write the name of a new process tree session to standard output, and then fork a background instance of a new `ttsession` to manage this new session.
`-s`	Silent. Do not write any warning messages to standard error.

Note — If neither the -c, -d, or -p options are specified, `ttsession` starts an `X` session for the display specified in the `$DISPLAY` environment variable.

Table 2-1 ttsession Command Line Options *(Continued)*

Argument	Description
-S	Do not fork a background instance to manage the ttsession session.
-t	Turn on trace mode. If trace mode is turned on while ttsession is running, messages appear on the console. Tracing displays the state of a message when it is first seen by ttsession. The lifetime of the message is then shown by showing the result of matching the message against type signatures (dispatch stage) and then showing the result of matching the message against any registered message patterns (delivery stage). Any attempt to send the message to a given process is shown along with the success of that attempt.
-v	Write the version number to standard output and exit.
-X	Read in the types from the following XDR format databases: $HOME/.tt/types.xdr <implementation-specific system and network databases> /usr/dt/appconfig/tttypes/types.xdr The database are listed in order of descending precedence. Entries in $HOME/.tt/types.xdr override any like entries in the database lower in the list. These databases can be overridden by setting the TTPATH environment variable.

Note — If neither the -c, -d, or -p options are specified, ttsession starts an X session for the display specified in the $DISPLAY environment variable.

ttsession responds to two signals.

- If it receives the SIGUSR1 signal, it toggles the trace mode on or off.
- If it receives the SIGUSR2 signal, it rereads the types file.

Background and Batch Sessions

Run your application as its own session if it runs as a background job, in a batch session, or in a session bound to a character terminal. To run your application in its own session, use the -c parameter with the ttsession command, as follows:

```
ttsession -c [ command-to-run-in-batch ]
```

This command will fork off a shell from which you can run your application.

Note – The `-c` parameter must be the last option on the command line; any characters placed after the `-c` parameter on the command line are taken as the command to be executed.

X Window System

To establish a session under the X Window System, execute `ttsession` either without arguments (which takes the display from the `$DISPLAY` environment variable) or specify the display with the `-d` parameter as follows:

```
ttsession -d :0
```

When `ttsession` is invoked, it immediately forks and the parent copy exits; the process managing the session executes in the background. The session is registered as a property, named by `_TT_SESSION` on the root window of screen 0; the host and port number is given for communication with the process managing the session.

Locating ttsession

To display the sessid of the session for the Xdisplay:

```
xprop -root | grep _TT_SESSION
```

Maintaining ToolTalk Files and Databases

The ToolTalk package contains a special set of shell commands you can use to copy, move, and remove ToolTalk files (that is, files mentioned in messages and files that contain ToolTalk objects). After a standard shell command (such as `cp`, `mv`, or `rm`) is performed, the ToolTalk service is notified that a file location has changed.

The ToolTalk package also contains a database check and repair utility for the ToolTalk database, `ttdbck`, that you can use to check and repair your ToolTalk databases.

Demonstration Programs

The ToolTalk service files contain the following demonstration program:

- ttsnoop

  A demonstration program that allows you to create and send custom-constructed ToolTalk messages. You can also use this program to selectively monitor any or all ToolTalk messages on your system.

Setting Up and Maintaining the ToolTalk Processes

3

Note – The ToolTalk database server program must be installed on all machines storing files that contain ToolTalk objects or files that are the subject of ToolTalk messages.

Location of the ToolTalk Service Files

This section lists the ToolTalk directories and files.

- `ttsession` communicates with other `ttsessions` on the network to deliver messages. `rpc.ttdbserverd` stores and manages ToolTalk object specs and information on files referenced in ToolTalk messages.

```
/usr/dt/bin/
  ttsession
  rpc.ttdbserverd
```

- These commands are standard operating system shell commands that inform the ToolTalk service when files containing ToolTalk objects or files that are the subject of ToolTalk messages are copied, moved, or removed.

```
/usr/dt/bin/
  ttcp
  ttmv
  ttrm
  ttrmdir
  ttar
```

- `ttdbck` is a database check and recovery tool for the ToolTalk databases.

  ```
  /usr/dt/bin/
   ttdbck
  ```

- `tt_type_comp` is a compiler for ptypes and otypes. It compiles the ptype and otype files and automatically installs them in the ToolTalk Types database.

  ```
  /usr/dt/bin/
   tt_type_comp
  ```

- These files are the application programming interface (API) libraries and header file that contain the ToolTalk functions used by applications to send and receive messages.

  ```
  /usr/dt/lib/
   libtt.sl
   libtt.a
  ```

  ```
  /usr/dt/include/tt
   tt_c.h
   tttk.h
  ```

- This is the location of working files and compiled databases.

  ```
  /TT_DB
  ```

- This is the ptypes database.

  ```
  /usr/dt/appconfig/tttypes/types.xdr
  ```

- This is the location of personal modifications to the ptypes database.

  ```
  /$HOME/.tt/*.xdr
  ```

- These are the man pages for the ToolTalk binary files, type compiler, enhanced shell commands, API, and database check utility.

  ```
  /usr/dt/man/man1
   tt_type_comp.1
   ttcp.1
   ttmv.1
   ttrm.1
   ttrmdir.1
   ttsession.1
   tttar.1
   tttrace.1
  ```

```
/usr/dt/man/man1m
 rpc.ttdbserverd.1m
 ttdbck.1m
```

```
/usr/dt/include/tt
(See Chapter 17, "ToolTalk Functions/Commands" for a list of the
ToolTalk message man pages)
```

```
/usr/dt/include/tt
(See Chapter 17, "ToolTalk Functions/Commands" for a list of the
ToolTalk message man pages)
```

```
/usr/dt/man/man5
 Tttt_c.5
 Ttttttk.5
```

```
/usr/dt/man/man6
 ttsnoop.6
```

Version

All ToolTalk commands support a `-v` option that prints the version string.

Environment Variables

ToolTalk Environment Variables

There are several ToolTalk environment variables that may be set:

- `TTSESSION_CMD`
- `TT_ARG_TRACE_WIDTH`
- `TT_FILE`
- `TT_HOSTNAME_MAP`
- `TT_PARTITION_MAP`
- `TT_SESSION`
- `TT_TOKEN`
- `TTPATH`
- `DISPLAY`

Table 3-1 describes these variables.

Table 3-1 Environment Variables

Variable	Description
`TTSESSION_CMD`	Overrides the standard options specified when tools automatically start ttsession. If this variable is set, all ToolTalk clients use this command to automatically start their `X` sessions.
`TT_ARG_TRACE_WIDTH`	Defines the number of characters of argument and context values to print when in trace mode. The default is to print the first 40 characters.
`TT_FILE`	`ttsession` places a pathname in this variable when a tool is invoked by a message scoped to the defined file.
`TT_HOSTNAME_MAP`	Points to a map file. The defined map file is read into the ToolTalk client for redirecting host machines.
`TT_PARTITION_MAP`	Points to a map file. The defined map file is read into the ToolTalk client for redirecting file partitions.
`TT_SESSION`	`ttsession` communicates its session identifier to the tools that it starts. If this variable is set, the ToolTalk client library uses its value as the default session identifier. **NOTE -** The string stored in this variable can be passed to `tt_default_session_set`.
`TT_TOKEN`	Notifies the ToolTalk client library that it has been started by ttsession; the client can then confirm to ttsession that the start was successful.
`TTPATH`	Tells the ToolTalk service where the ToolTalk Types databases reside. The format of this variable is: `userDB[:systemDB[:networkDB]]`
`DISPLAY`	Causes ttsession to communicate its session identifier to the tools that it starts if the `TT_SESSION` variable is not set. If the `DISPLAY` variable is set, the ToolTalk client library uses its value as the default session identifier.

A process is given a modified environment when it is automatically started by the ToolTalk service. The modified environment includes the environment variables `$TT_SESSION`, `$TT_TOKEN`, and any contexts in the start-message whose keyword begins with the dollar sign symbol ($). Optionally, the environment variable $TT_FILE may also be included in the modified environment if it is a file-scoped message.

Note – If the tt_open call will be invoked by a child process, the parent process must propagate the modified environment to that child process.

Other Environment Variables

The TMPDIR environment variable is another environment variable that you can set to manipulate the ToolTalk development environment; for example:

```
TMPDIR=/var/tmp
```

redirects files to the /var/tmp directory.

Using Context Slots to Create Environment Variables

Message contexts have a special meaning when the ToolTalk service starts an application. If the name of a context slot begins with a dollar sign ($), the ToolTalk service interprets the value as an environment variable: for example,

```
start "my_application $CON1"
```

uses the value of context slot $CON1.

Installing the ToolTalk Database Server

The ToolTalk Database server is used to store three types of information:

1. ToolTalk objects specs.
2. ToolTalk session ids of sessions with clients that have joined a file using the tt_file_join call.
3. File-scoped messages that are queued because the message disposition is TT_QUEUED and a handler that can handle the message has not yet been started.

Note – The ToolTalk database server does *not* store messages that are scoped to file-in-session.

The ToolTalk service requires that a database server run on each machine that stores files that contain ToolTalk objects or files that are the subject of ToolTalk messages. When an application attempts to reference a file on a machine that does not contain a database server, an error similar to the following message is displayed:

```
% Error: Tool Talk database server on integral is not running: tcp
```

where *integral* is the hostname and *tcp* is the application protocol. This error message indicates that the connection failed. A failed connection can also be caused by network problems.

When the ToolTalk Service is Installed Elsewhere on the System

To install the ToolTalk database server from another machine on the system that already has the ToolTalk service installed:

1. Login as superuser.
2. Check that the `/etc/inetd.conf` file contains the following line.

```
# rpc stream tcp swait root /usr/dt/bin/rpc.ttdbserverd 100083 1
/usr/dt/bin/rpc.ttdbserverd
```

3. Cause `inetd` to reread the configuration file.

```
# /etc/inetd -c
```

Running the New ToolTalk Database Server

Once the new version of the ToolTalk database server has been run on a machine, you cannot revert to a previous version of the ToolTalk database server. Any attempt to run a previous version of the ToolTalk database server displays the following error message:

```
rpc.ttdbserverd: Any data written using a ToolTalk 1.0.x DB server
after using a new ToolTalk DB server will be ignored.
```

Redirecting the ToolTalk Database Server

You can redirect both the database host machines and the file system partitions.

- Redirecting a database host machine allows a ToolTalk client to physically access ToolTalk data from a machine that is not running a ToolTalk database server.
- Redirecting a file system partition allows a ToolTalk database to logically read and write ToolTalk data from and to a read-only file system partition (for example, a CD-ROM) by physically accessing a different file system partition.

Redirecting the Host Machine

When you redirect a database host machine, a ToolTalk client can physically access ToolTalk data from a machine that is not running a ToolTalk database server. To redirect the host machine, you need to map the hostnames of the machines the ToolTalk client is to access. On the machine running the ToolTalk client that is making the database query:

1. Create a `hostname_map` file.

   For example:

```
# Map first host machine
oldhostname1  newhostname1

# Map second host machine
oldhostname2  newhostname2
```

   where *oldhostname* is the name of the machine the ToolTalk client needs to access and *newhostname* is the name of a machine that is running the ToolTalk database server.

2. If you want to make changes or add types on a system-wide basis, use `/etc/dt/appconfig/tttypes`.

Note – A file defined in the `TT_HOSTNAME_MAP` environment variable has a higher precedence than the map in the *user* database.

The map file is read into a ToolTalk client when the client makes a `tt_open` call.

Redirecting the File System Partition

When you redirect a file system partition, a ToolTalk database can logically read and write ToolTalk data from and to a read-only file system partition by physically accessing a different file system partition. To redirect a file partition, you need to map the partitions to where the ToolTalk database will write. On the machine running the ToolTalk database server:

1. Create a `partition_map` file.

   For example:

```
# Map first partition
/cdrom   /usr
```

   maps the read-only partition `/cdrom` to `/usr`, a read-write partition.

2. Store the map file in the same location at which the system ToolTalk Types databases are stored.

Note – A file partition defined in the `TT_PARTITION_MAP` environment variable has a higher precedence than the file partition defined in this map file.

The map file is read when the ToolTalk database server is started, or when the database receives a `USR2` signal.

Maintaining Application Information 4

Applications that want to receive ToolTalk messages, provide information to the ToolTalk service that describes what kind of messages they want to receive. This information, known as message patterns, is provided dynamically either by applications as they run, or through ptype and otype files.

Installing Application Types

Installing application types is an occasional task; you only need to install type information when an application error condition exists. Ptype and otype files are run through the ToolTalk type compiler at installation time. `tt_type_comp` merges the information into the ToolTalk Types Database. The application then tells the ToolTalk service to read the type information in the database.

To install an application's ptype and otype files, follow these steps:

1. Run `tt_type_comp` on your type file.

```
% tt_type_comp <your-file>
```

   `tt_type_comp` runs *your-file* through `cpp`, compiles the type definitions, and merges the information into a ToolTalk Types table. Table 4-1 describes location of the XDR-base format tables.

Table 4-1 XDR-base Format ToolTalk Types Tables

Database	Uses XDR Table
user	`~/.tt/types.xdr`
system	`/etc/dt/appconfig/tttypes/types .xdr`

By default, `tt_type_comp` uses the *user* database. To specify another database, use the `-d` option; for example:

```
% tt_type_comp -d user|system|network <your-file>
```

Note – When you run `tt_type_comp` on your ptype or otype files, it first runs `cpp` on the file and then checks the syntax before it places the data into the ToolTalk Types Database format. If syntax errors are found, a message is displayed that indicates the line number of the cpp file. To find the line, enter

`cpp -P` *source-file temp-file*

and view the *temp-file* to find the error on the line reported by `tt_type_comp`.

---

2. Force `ttsession` to reread the ToolTalk Types Database.

   To force `ttsession` to reread the ToolTalk Types Database, see "Updating the ToolTalk Service" on page 29.

Examining ToolTalk Type Information

You can examine all type information in a specified ToolTalk Types Database, only the ptype information, or only the otype information. To specify the database you want to examine, use the `-d` option and supply the name of the *user*, *system*, or *network* to indicate the desired database. If the `-d` option is not used, `tt_type_comp` will use the *user* database by default.

♦ To examine all the ToolTalk type information in a ToolTalk Types Database, enter the following line

```
% tt_type_comp -d user|system|network -p
```

The type information will be printed out in source format.

♦ To list all ptypes in a ToolTalk Types Database, enter the following line:

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

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

♦ To list all otypes in a ToolTalk Types Database, enter the following line:

```
% tt_type_comp -d user|system|network -O
```

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

Removing ToolTalk Type Information

You can remove both ptype and otype information from the ToolTalk Types Databases.

♦ Use `tt_type_comp` to remove type information. Enter the following line:

```
% tt_type_comp -d user|system|network -r type
```

For example, to remove a ptype called *EditDemo* from the ToolTalk Types network database of a sample application, enter the line:

```
% tt_type_comp -d network -r EditDemo
```

After you remove type information, force any running `ttsessions` to reread the ToolTalk Types Database again to bring the ToolTalk service up-to-date. For more information, see “Updating the ToolTalk Service” on page 29.

Updating the ToolTalk Service

When you make changes to the ToolTalk Types Database, you must force any ToolTalk sessions that are already running to reread it or the changes will not be in effect. Use `ttsession` to force the ToolTalk service to read the type information.

1. Enter the `ps` command to find the process identifier (*pid*) of the `ttsession` process

```
% ps -ef | grep ttsession
```

2. Enter the `kill` command to send a SIGUSR2 signal to `ttsession`.

```
% kill -USR2 <ttsession pid>
```

Process Type Errors

One or both of the following conditions exists if applications report the error:

`Application is not an installed ptype.`

1. The ToolTalk service has not been instructed by the application to reread the recently updated type information in the ToolTalk Types Database. For instructions on how to force the ToolTalk service to reread type information from the ToolTalk Types Database, see "Updating the ToolTalk Service" on page 29.
2. The application's ptypes and otypes have not been compiled and merged into the ToolTalk Types Database. For instructions on how to compile and merge type information, see "Installing Application Types" on page 27.

Using TTSnoop to Debug Messages and Patterns

TTSnoop is a tool that creates and sends custom-constructed ToolTalk messages. You can also use TTSnoop as a tool to selectively monitor any or all ToolTalk messages.

About TTSnoop

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

TTSnoop can also be used for:

- creating, receiving and destroying messages.
- creating, opening, and destroying patterns.
- joining a session.
- scoping to a file and generating a list of ToolTalk-based desktop actions.

How to Run TTSnoop

To run TTSnoop, execute the following command:

```
/usr/dt/bin/ttsnoop
```

TTSnoop has several command line options. See the `ttsnoop(6)` man page for details.

How to Turn Tracing On and Off

Tracing is turned on by default. To turn Tracing off, do the following:

♦ From the Snoop menu, toggle the On/Off item.

How to Create a Message

To create a ttmedia Edit request message, do the following:

♦ From the Sessions menu, select Join...

3. Push the Default Session button in the tt_session_join window, then push the Join button.
4. From the Message menu, select Create...
5. Make the following selections in the Create window:
   - Class: REQUEST
   - Address: PROCEDURE
   - Scope: SESSION
   - Op: Edit
   - Disposition: DISCARD
   - File: <enter file name - the file does not have to exist>
   - Push the Add Arg... button and make these selections:
     1) Mode: INOUT; 2) vtype: unknown; 3) push the Add button

▼ *To Send the Message You Have Created*

♦ Push the Message button and select Send.

How to Generate a List of ToolTalk Actions

To generate a list of ToolTalk actions, do the following:

♦ From the Types menu, select TTMsg Actions...

Maintaining Files and Objects Referenced in ToolTalk Messages 5

ToolTalk messages can reference files of interest or ToolTalk objects. The ToolTalk service maintains information about files and objects, and needs to be informed of changes to these files or objects.

The ToolTalk service provides wrapped shell commands to move, copy, and remove files. These commands inform the ToolTalk service of any changes.

ToolTalk-Enhanced Shell Commands

The ToolTalk-enhanced shell commands described in Table 5-1 first invoke the standard shell commands with which they are associated (for example, `ttmv` invokes `mv`) and then update the ToolTalk service with the file changes. It is necessary to use the ToolTalk-enhanced shell commands when working with files that contain ToolTalk objects.

Table 5-1 ToolTalk-Enhanced Shell Commands

Command	**Definition**	**Syntax**
`ttcp`	Copies files that contain objects.	`ttcp` *source-file destination-file*
`ttmv`	Renames files that contain objects.	`ttmv` *old new*

Table 5-1 ToolTalk-Enhanced Shell Commands *(Continued)*

Command	Definition	Syntax
`ttrm`	Removes files that contain objects.	`ttrm` *file*
`ttrmdir`	Removes empty directories that are associated with ToolTalk objects. You also create an object spec for a directory. For example: if a directory is mentioned in a file-scoped message, when an object spec is created, the path name of a file or directory is supplied.	`ttrmdir` *directory*
`tttar`	Archives and de-archives files that contain ToolTalk objects.	`tttar c|t|x` *pathname1 pathname2*

You can cause the ToolTalk-enhanced shell commands to be executed when the standard shell commands are invoked. To do this, alias the ToolTalk-enhanced shell commands in the shell startup file so that the enhanced commands appear as standard shell commands.

```
# ToolTalk-aware shell commands in .cshrc
alias mvttmv
alias cpttcp
alias rmttrm
alias rmdirttrmdir
alias tartttar
```

Maintaining and Updating ToolTalk Databases

Information about files and objects in the ToolTalk databases can become outdated if the ToolTalk-enhanced shell commands are not used to copy, move, and remove them. For example, you can remove a file *old_file* that contains ToolTalk objects from the file system with the standard `rm` command; however, because the standard shell command does not inform the ToolTalk service that *old_file* has been removed, the information about the file and the individual objects remains in the ToolTalk database.

To remove the file and object information from the ToolTalk database, use the command:

`ttrm -L` *old_file*

Displaying, Checking, and Repairing Databases

Use the ToolTalk database utility `ttdbck` to display, check, or repair ToolTalk databases. You also use the `ttdbck` utility for operations such as:

- Removing all ToolTalk objects of a given otype; for example, an otype that has been de-installed.
- Moving specific ToolTalk objects from one file to another.
- Searching for all ToolTalk objects that reference nonexistent files.

Note – ToolTalk databases are typically accessible only to root; therefore, the `ttdbck` utility is normally run as root.

Participating in ToolTalk Sessions 6

This chapter provides instructions on how to participate in a ToolTalk session. It also shows you how to manage the storage of values passed in from the ToolTalk service and how to handle errors that the ToolTalk service returns.

To use the ToolTalk service, your application calls ToolTalk functions from the ToolTalk application programming interface (API) library. To modify your application to use the ToolTalk service, you must first include the ToolTalk API header file in your program. After you have initialized the ToolTalk service and joined a session, you can join files and additional user sessions. When your process is ready to quit, you unregister your message patterns and leave your ToolTalk session.

The ToolTalk Libraries

The ToolTalk libraries are located in the following directory: `/usr/dt/lib`. The name of the archived library is `libtt` and its extension may vary slightly between platforms. However, the prefix ofthe name will remain the same.

Including the ToolTalk API Header File

To modify your application to use the ToolTalk service, you must first include the ToolTalk API header file `tt_c.h` in your program. This file resides in the `/usr/dt/include/Tt` directory.

The following code sample shows how a program includes this file.

```
#include <stdio.h>
#include <sys/param.h>
#include <sys/types.h>

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

Registering with the ToolTalk Service

Before you can participate in ToolTalk sessions, you must register your process with the ToolTalk service. You can either register in the ToolTalk session in which the application was started (the *initial session*), or locate another session and register there.

The ToolTalk functions you need to register with the ToolTalk service are shown in Table 6-1.

Table 6-1 Registering with the ToolTalk Service

Return Type	ToolTalk Function	Description
`char *`	`tt_open(void)`	Process identifier.
`int`	`tt_fd(void)`	File descriptor.
`char *`	`tt_X_session(const char *xdisplay)`	Return the session identifier of the specified `X` display server.
`Tt_status`	`tt_default_session_set(const char *sessid)`	Sets the session to which `tt_open` will connect.

Registering in the Initial Session

To initialize and register your process with the initial ToolTalk session, your application needs to obtain a process identifier (*procid*). You can then obtain the file descriptor (*fd*) that corresponds to the newly initialized ToolTalk process.

The following code sample first initializes and registers the sample program with the ToolTalk service, and then obtains the corresponding file descriptor.

```
int ttfd;
```

```
char    *my_procid;

/*
 * Initialize ToolTalk, using the initial default session
 */

my_procid = tt_open();

/*
 * obtain the file descriptor that will become active whenever
 * ToolTalk has a message for this process.
 */

ttfd = tt_fd();
```

`tt_open` returns the procid for your process and sets it as the default procid; `tt_fd` returns a file descriptor for your current procid that will become active when a message arrives for your application.

Caution – Your application must call `tt_open` before other `tt_` calls are made; otherwise, errors may occur. There are, however, some exceptions, such as: `tt_default_session_set`, `tt_default_procid`, `tt_X_session`, `tt_netfile_file`, `tt_file_netfile`, `tt_host_file_netfile`, and `tt_host_netfile_file`.

Note – Transparent to an application, the TED action and data typing services use ToolTalk to execute message-type actions and to notify an application when the user has changed an action or data typing database. To do this, these services automatically register an application with a ToolTalk session if the application is not registered, or the service will reuse the procid if the application is already registered. In general, an application will only want to register with one ToolTalk session. Consequently, it is recommended that an application call `tt_open` before it uses `libDtSvc` service functions such as `DtDbLoad`, `DtActionInvoke`, or `DtDbReloadNotify`.

When `tt_open` is the first call made to the ToolTalk service, it sets the initial session as the default session. The default session identifier (*sessid*) is important to the delivery of ToolTalk messages. The ToolTalk service automatically fills in the default sessid if an application does not explicitly set the session message attribute. If the message is scoped to `TT_SESSION`, the message will be delivered to all applications in the default session that have registered interest in this type of message.

Registering in a Specified Session

To register in a session other than the initial session, your program must find the name of the other session, set the new session as the default, and register with the ToolTalk service.

The following code sample shows how to join an `X` session named `somehost:0` that is not your initial session.

```
char   *my_session;
char   *my_procid;

my_session = tt_X_session("somehost:0");
tt_default_session_set(my_session);
my_procid = tt_open();
ttfd = tt_fd();
```

Note – The required calls must be in the specified order.

1. `tt_X_session();`

   This call retrieves the name of the session associated with an `X` display server. `tt_X_session()` takes the argument

   ```
   char *xdisplay_name
   ```

   where `xdisplay_name` is the name of an X display server (in this example, `somehost:0`, `:0`).

2. `tt_default_session_set();`

   This call sets the new session as the default session.

3. `tt_open();`

   This call returns the procid for your process and sets it as the default procid.

4. `tt_fd();`

   This call returns a file descriptor for your current procid.

Registering in Multiple Sessions

There may be cases when you want to send and receive your messages in different sessions. To register in multiple sessions, your program must find the identifiers of the sessions to which it wants to connect, set the new sessions, and register with the ToolTalk service.

The following code sample shows how to connect *procid* to *sessid1*, and *procid2* to *sessid2*.

```
tt_default_session_set(sessid1);
my_procid1 = tt_open();
tt_default_session_set(sessid2);
my_procid2 = tt_open();
tt_fd2 = tt_fd();
```

You can then use `tt_default_procid_set` to switch between the sessions.

Setting Up to Receive Messages

Before your application can receive messages from other applications, you must set up your process to watch for arriving messages. When a message arrives for your application, the file descriptor becomes active. The code you use to alert your application that the file descriptor is active depends on how your application is structured.

For example, a program can have a callback function invoked when the file descriptor becomes active. The following code sample invokes `notify_set_input_func` with the handle for the message object as a parameter.

```
/*
 * Arrange for a program to call receive_tt_message when the
 * ToolTalk file descriptor becomes active.
 */
notify_set_input_func(base_frame,
                          (Notify_func)receive_tt_message,
                      ttfd);
```

Table 6-2 describes various window toolkits and the call used to watch for arriving messages.

Table 6-2 Code Used to Watch for Arriving Messages

Window Toolkits	Code Used
XView	`notify_set_input_func()`
X Window System Xt (Intrinsics)	`XtAddInput()` or `XtAddAppInput()`
Other toolkits including Xlib structured around `select(2)` or `poll(2)` system calls	The file descriptor returned by `tt_fd()` Note - Once the file descriptor is active and the select call exits, use `tt_message_receive()` to obtain a handle for the incoming message.

Sending and Receiving Messages in the Same Process

Normally, the receiver deletes the message when it has completed the requested operation; however, the ToolTalk service uses the same message ID for both the receiver and the requestor. When sending and receiving messages in the same process, these features cause the message underneath the requestor to be deleted as well.

One workaround is to put a refcount on the message. To do this, use the `tt_message_user[_set]()` function.

Another workaround is to destroy the message in the receiver only if the sender is not the current procid; for example:

```
Tt_callback_action
my_pattern_callback(Tt_message m, Tt_pattern p)
{
    /* normal message processing goes here */

    if (0!=strcmp(tt_message_sender(m),tt_default_procid()) {
        tt_message_destroy(m);
    }
    return TT_CALLBACK_PROCESSED;
}
```

Sending and Receiving Messages in a Networked Environment

You can use the ToolTalk service in a networked environment; for example, you can start a tool on a different machine or join a session that is running on a different machine. To do this, invoke a ttsession with either the `-c` or `-p` option.

- The `-c` option will invoke the named program and place the right session id in its `TT_SESSION` environment variable. For example, the command

```
ttsession -c cmdtool
```

  defines `TT_SESSION` in that `cmdtool` and any ToolTalk client you run with the environment variable `$TT_SESSION` set to its value will join the session owned by this `ttsession`.

- The `-p` option prints the session id to standard output. `ttsession` then forks into the background to run that session.

To join the session, an application must either pass the session id to `tt_default_session_set` or place the session id in the environment variable `TT_SESSION` before it calls the `tt_open` function. `tt_open` will check the environment variable, `TT_SESSION`, and join the indicated session (if it has a value).

Unregistering from the ToolTalk Service

When you want to stop interacting with the ToolTalk service and other ToolTalk session participants, you must *unregister* your process before your application exits.

```
    /*
     * Before leaving, allow ToolTalk to clean up.
     */
    tt_close();

    exit(0);
}
```

`tt_close` returns `Tt_status` and closes the current default procid.

Sending Messages 7

This chapter explains how messages are routed, and describes the ToolTalk message attributes and algorithm. It also describes how to create messages, fill in message contents, attach callbacks to requests, and send messages.

How the ToolTalk Service Routes Messages

Applications can send two classes of ToolTalk messages: *notices* and *requests.* A notice is informational, a way for an application to announce an event. Applications that receive a notice absorb the message without returning results to the sender. A request is a call for an action, with the results of the action recorded in the message, and the message returned to the sender as a reply.

Sending Notices

When you send an informational message, the notice takes a one-way trip, as shown in Figure 7-1.

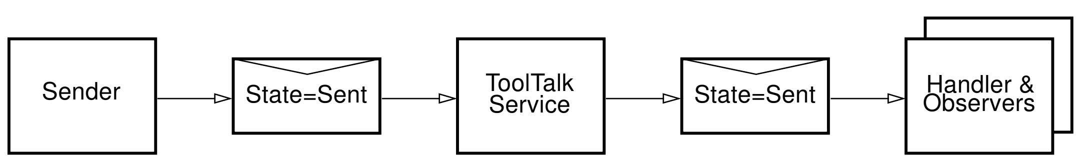

Figure 7-1 Notice Routing

The sending process creates a message, fills in attribute values, and sends it. The ToolTalk service matches message and pattern attribute values, then gives a copy of the message to one handler and to all matching observers. File-scoped messages are automatically transferred across session boundaries to processes that have declared interest in the file.

Sending Requests

When you send a message that is a request, the request takes a round-trip from sender to handler and back; copies of the message take a one-way trip to interested observers. Figure 7-2 illustrates the request routing procedure.

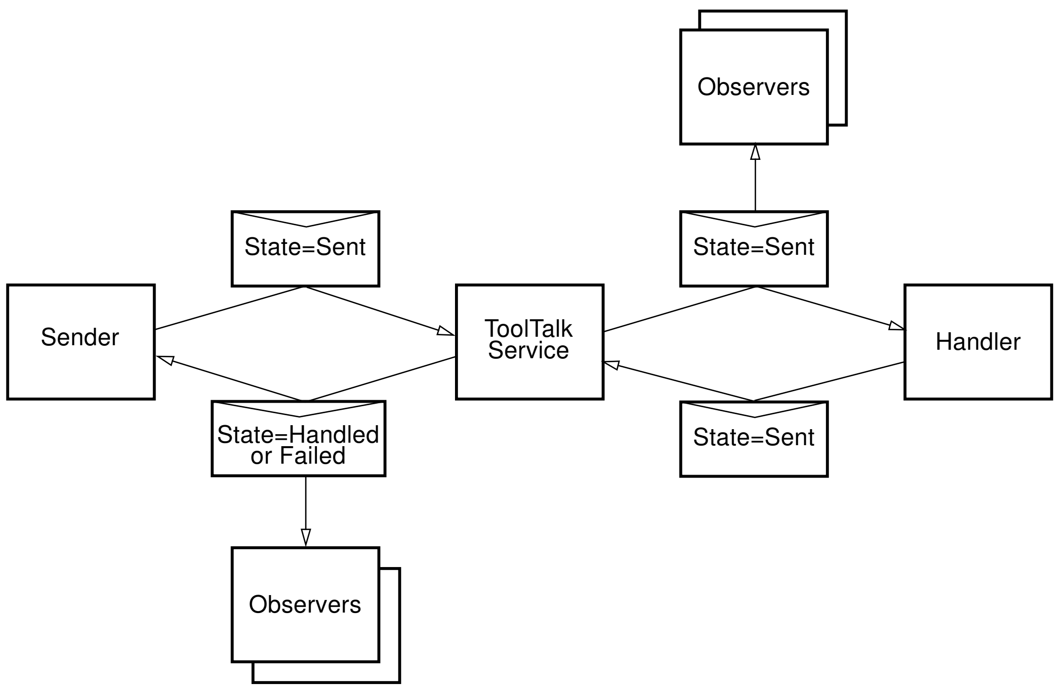

Figure 7-2 Request Routing

The ToolTalk service delivers a request to only one handler. The handler adds results to the message and sends it back. Other processes can observe a request before or after it is handled, or at both times; observers absorb a request without sending it back.

Changes in State of Sent Message

To allow you to track the progress of a request you sent, you will receive a message every time the request changes state. You will receive these state-change messages even if no patterns have been registered, or no message callbacks have been specified.

Message Attributes

ToolTalk messages contain attributes that store message information and provide delivery information to the ToolTalk service. This delivery information is used to route the messages to the appropriate receivers.

ToolTalk messages are simple structures that contain attributes for address, subject (such as *operation* and *arguments*), and delivery information (such as *class* and *scope*). Each message contains attributes from Table 7-1.

Table 7-1 ToolTalk Message Attributes

Message Attribute	Value	Description	Who Can Complete
Arguments	arguments or results	Specifies arguments used in the operation. If the message is a reply, these arguments contain the results of the operation.	Sender, receiver
Class	`TT_NOTICE`, `TT_REQUEST`	Specifies whether the recipient needs to perform an operation.	Sender
File	`char *pathname`	Specifies the file involved in the operation. If the scope of the message does not require a file, the file is an attribute only.	Sender, ToolTalk
Object	`char *objid`	Specifies the object involved in the operation.	Sender, ToolTalk
Operation	`char *opname`	Specifies the name of operation to be performed.	Sender
Otype	`char *otype`	Specifies the type of object involved in the operation.	Sender, ToolTalk

Table 7-1 ToolTalk Message Attributes *(Continued)*

Message Attribute	Value	Description	Who Can Complete
Address	`TT_PROCEDURE, TT_OBJECT, TT_HANDLER, TT_OTYPE`	Specifies where the message should be sent.	Sender
Handler	`char *procid`	Specifies the receiving process.	Sender, ToolTalk
Handler_ptype	`char *ptype`	Specifies the type of receiving process.	Sender, ToolTalk
Disposition	`TT_DISCARD, TT_QUEUE, TT_START TT_START+TT_QUEUE`	Specifies what to do if the message cannot be received by any running process.	Sender, ToolTalk
Scope	`TT_SESSION, TT_FILE, TT_BOTH, TT_FILE_IN_SESSION`	Specifies the applications that will be considered as potential recipients based on their registered interest in a session or file.	Sender, ToolTalk
Sender_ptype	`char *ptype`	Specifies the type of the sending process.	Sender, ToolTalk
Session	`char *sessid`	Specifies the sending process' session.	Sender, ToolTalk
Status	`int status, char *status_str`	Specifies additional information about the status of the message.	Receiver, ToolTalk

Address Attribute

Messages addressed to other applications can be addressed to a particular process or to any process that has registered a pattern that matches your message. When you address a message to a process, you need to know the process identifier (*procid*) of the other application; however, processes do not usually know each other's *procid*. More often, a sender does not care which process performs an operation (request message) or learns of an event (notice message).

Scope Attributes

Applications that use the ToolTalk service to communicate usually have something in common – the applications are running in the same session, or they are interested in the same file or data. To register this interest, applications join sessions or files (or both) with the ToolTalk service. This file and session information is used by the ToolTalk service with the message patterns to determine which applications should receive a message.

File Scope

When a message is scoped to a file, only those applications that have joined the file (and match the remaining attributes) will receive the message. Applications that share interest in a file do not have to be running in the same session.

File-based Scoping in Patterns

Table 7-2 describes the types of scopes that use files that you can use to scope messages with patterns.

Table 7-2 Scoping a Message with Patterns to a File

Type of Scope	Description
`TT_FILE`	Scopes to the specified file only. You can set a session attribute on this type of pattern to provide a file-in-session-like scoping but a `tt_session_join` call will *not* update the session attribute of a pattern that is scoped to `TT_FILE`.
`TT_BOTH`	Scopes to the *union* of interest in the file and the session. A pattern with only this scope will match messages that are scoped to the file, or scoped to the session, or scoped to both the file *and* the session.
`TT_FILE_IN_SESSION`	Scopes to the *intersection* of interest in the file and the session. A pattern with only this scope will *only* match messages that are scoped to both the file and session.

To scope to the union of TT_FILE_IN_SESSION and TT_SESSION, add both scopes to the same pattern, as shown in the following example:

```
tt_open();

Tt_pattern pat = tt_create_pattern();
tt_pattern_scope_add(pat, TT_FILE_IN_SESSION);
tt_pattern_scope_add(pat, TT_SESSION);
tt_pattern_file_add(pat, file);
tt_pattern_session_add(pat, tt_default_session());
tt_pattern_register(pat);
```

File-based Scoping in Messages

Messages have the same types of file-based scoping mechanisms as patterns. Table 7-3 describes these scopes.

Table 7-3 Scoping Mechanisms for Messages

Type of Scope	Description
`TT_FILE`	Scopes the message to all clients that have registered interest in a file.
`TT_BOTH`	Scopes the message to all clients that have registered interest in the message's session, the message's file, or the message's session and file.
`TT_FILE_IN_SESSION`	Scopes the message to all clients that have registered interest in both the message's file and session.
`TT_SESSION +` `tt_message_file_set()`	Scopes the message to every client that has registered interest in the message's session. When the message is received by a client whose pattern matches, the receiving client can call `tt_message_file` to get the file name.

When a message is scoped to TT_FILE or TT_BOTH, the ToolTalk client library checks the database server for all sessions that have clients that are interested in the file and sends the message to all of the interested ToolTalk sessions. The ToolTalk sessions then match the messages to the appropriate clients. The message sender is *not* required to explicitly call to tt_file_join.

If a message that is scoped to TT_FILE_IN_SESSION or TT_SESSION contains a file, the database server is not contacted and the message is sent only to clients that are scoped to the message's session.

Session Scope

When a message is scoped to a session, only those applications that have connected to that session are considered as potential recipients. Note the following example:

```
/*Create message*/
Tt_message m= tt_message_create();

/*Add scope to message*/
tt_message_scope_set(m, TT_SESSION);

/*Add file attribute that does not affect message scope*/
tt_message_file_set(m, file);
```

File-In-Session Scope

Applications can be very specific about the distribution of a message by specifying `TT_FILE_IN_SESSION` for the message scope. Only those applications that have joined both the file and the session indicated are considered potential recipients.

Applications can also scope a message to every client that has registered interest in the message's session by specifying `TT_SESSION` with `tt_message_file_set` for the message scope. When the message is received by a client whose pattern matches, the receiving client can get the file name by calling `tt_message_file`. Here is an example of setting a file:

```
/*Create message*/
Tt_message m= tt_message_create();

/*Add scope to message*/
tt_message_scope_set(m, TT_FILE_IN_SESSION);

/*Add file to message scope*/
tt_message_file_set(m, file);
```

Serialization of Structured Data

The ToolTalk service supports three types of data for message arguments: integers, null-terminated strings, and byte strings.

To send any other data type in a ToolTalk message, the client must serialize the data into a string or byte string and then deserialize it on receipt. The new XDR-argument API calls provided with the ToolTalk service now handles these serialization and deserialization functions. The client only needs to provide an XDR routine and a pointer to the data. After serializing the data into the internal buffer, the ToolTalk service treats the data in the same manner as it treats a byte stream.

ToolTalk Message Delivery Algorithm

To help you better understand how the ToolTalk service determines message recipients, this section describes the creation and delivery of both process-oriented messages and object-oriented messages.

Process-Oriented Message Delivery

For some process-oriented messages, the sending application knows the ptype or the procid of the process that should handle the message. For other messages, the ToolTalk service can determine the handler from the operation and arguments of the message.

1. Initialize.

   The sender obtains a message handle and fills in the *address*, *scope*, and *class* attributes.

   The sender fills in the *operation* and *arguments* attributes.

   If the sender has declared only one ptype, the ToolTalk service fills in *sender_ptype* by default; otherwise, the sender must fill it in.

   If the scope is `TT_FILE`, the file name must be filled in or defaulted. If the scope is `TT_SESSION`, the session name must be filled in or defaulted. If the scope is `TT_BOTH` or `TT_FILE_IN_SESSION`, both the file name and session name must be filled in or defaulted.

Note – The set of patterns checked for delivery depends on the scope of the message. If the scope is TT_SESSION, only patterns for processes in the same session are checked. If the scope is TT_FILE, patterns for all processes observing the file are checked. If the scope is TT_FILE_IN_SESSION or TT_BOTH, both sets of processes are checked.

The sender may fill in the *handler_ptype* if known; however, this greatly reduces flexibility because it does not allow processes of one ptype to substitute for another. Also, the disposition attribute must be specified by the sender in this case.

2. Dispatch to handler.

   The ToolTalk service compares the *address*, *scope*, *message class*, *operation*, and *argument* modes and types to all signatures in the Handle section of each ptype.

   Only one ptype will usually contain a message pattern that matches the operation and arguments and specifies a handle. If a handler ptype is found, then the ToolTalk service fills in *opnum*, *handler_ptype*, and *disposition* from the ptype message pattern.

   If the address is TT_HANDLER, the ToolTalk service looks for the specified procid and adds the message to the handler's message queue. TT_HANDLER messages cannot be observed because no pattern matching is done.

3. Dispatch to observers.

   The ToolTalk service compares the *scope*, *class*, *operation*, and *argument* types to all message patterns in the Observe section of each ptype.

   For all observe signatures that match the message and specify TT_QUEUE or TT_START, the ToolTalk service attaches a record (called an "observe promise") to the message that specifies the ptype and the queue or start options. The ToolTalk service then adds the ptype to its internal ObserverPtypeList.

4. Deliver to handler.

   If a running process has a registered handler message pattern that matches the message, the ToolTalk service delivers the message to the process; otherwise, the ToolTalk service honors the disposition (start or queue) options.

If more than one process has registered a dynamic pattern that matches the handler information, the more specific pattern (determined by counting the number of non-wildcard matches) is given preference. If two patterns are equally specific, the choice of handler is arbitrary.

5. Deliver to observers.

   The ToolTalk service delivers the message to all running processes that have registered Observer patterns that match the message. As each delivery is made, the ToolTalk service checks off any observe promise for the ptype of the observer. After this process is completed and there are observe promises left unfulfilled, the ToolTalk service honors the start and queue options in the promises.

Example

In this example, a debugger uses an editor to display the source around a breakpoint through ToolTalk messages.

The editor has the following Handle pattern in its ptype:

```
(HandlerPtype: TextEditor;
 Op: ShowLine;
 Scope: TT_SESSION;
 Session: my_session_id;
 File: /home/butterfly/astrid/src/ebe.c)
```

1. When the debugger reaches a breakpoint, it sends a message that contains the *op* (`ShowLine`), *argument* (the line number), *file* (the file name), *session* (the current session id), and *scope* (`TT_SESSION`) attributes.

2. The ToolTalk service matches this message against all registered patterns and finds the pattern registered by the editor.

3. The ToolTalk service delivers the message to the editor.

4. The editor then scrolls to the line indicated in the argument.

Object-Oriented Message Delivery

Many messages handled by the ToolTalk service are directed at objects but are actually delivered to the process that manages the object. The message signatures in an otype, which include the ptype of the process that can handle each specific message, help the ToolTalk service determine the process to which it should deliver an object-oriented message.

1. Initialize.

   The sender fills in the *class*, *operation*, *arguments*, and the target *objid* attributes.

   The sender attribute is automatically filled in by the ToolTalk service. The sender can either fill in the *sender_ptype* and *session* attributes or allow the ToolTalk service to fill in the default values.

   If the scope is `TT_FILE`, the file name must be filled in or defaulted. If the scope is `TT_SESSION`, the session name must be filled in or defaulted. If the scope is `TT_BOTH` or `TT_FILE_IN_SESSION`, both the file name and session name must be filled in or defaulted.

Note – The set of patterns checked for delivery depends on the scope of the message. If the scope is `TT_SESSION`, only patterns for processes in the same session are checked. If the scope is `TT_FILE`, patterns for all processes observing the file are checked. If the scope is `TT_FILE_IN_SESSION` or `TT_BOTH`, both sets of processes are checked.

---

2. Resolve.

   The ToolTalk service looks up the *objid* in the ToolTalk database and fills in the otype and file attributes.

3. Dispatch to handler.

   The ToolTalk service searches through the otype definitions for Handler message patterns that match the message's *operation* and *arguments* attributes. When a match is found, the ToolTalk service fills in *scope*, *opnum*, *handler_ptype*, and *disposition* from the otype message pattern.

4. Dispatch to object-oriented observers.

   The ToolTalk service compares the message's *class*, *operation*, and *argument* attributes against all Observe message patterns of the otype. When a match is found, if the message pattern specifies `TT_QUEUE` or `TT_START`, the ToolTalk service attaches a record (called an "observe promise") to the message that specifies the ptype and the queue or start options.

5. Dispatch to procedural observers.

   The ToolTalk service continues to match the message's *class*, *operation*, and *argument* attributes against all Observe message patterns of all ptypes. When a match is found, if the signature specifies `TT_QUEUE` or `TT_START`, the ToolTalk service attaches an observe promise record to the message, specifying the ptype and the queue or start options.

6. Deliver to handler.

   If a running process has a registered Handler pattern that matches the message, the ToolTalk service delivers the message to the process; otherwise, the ToolTalk service honors the disposition (queue or start) options.

   If more than one process has registered a dynamic pattern that matches the handler information, the more specific pattern (determined by counting the number of non-wildcard matches) is given preference. If two patterns are equally specific, the choice of handler is arbitrary.

7. Deliver to observers.

   The ToolTalk service delivers the message to all running processes that have registered Observer patterns that match the message. As each delivery is made, the ToolTalk service checks off any observe promise for the ptype of the observer. After this process is completed and there are observe promises left unfulfilled, the ToolTalk service honors the disposition (queue or start) options in the promises.

Example

In this example, a hypothetical spreadsheet application named FinnogaCalc is integrated with the ToolTalk service.

1. FinnogaCalc starts and registers with the ToolTalk service by declaring its ptype, `FinnogaCalc`, and joining its default session.

2. FinnogaCalc loads a worksheet, `hatsize.wks`, and tells the ToolTalk service it is observing the worksheet by joining the worksheet file.
3. A second instance of FinnogaCalc (called FinnogaCalc_2) starts, loads a worksheet, `wardrobe.wks`, and registers with the ToolTalk service in the same way.
4. The user assigns the value of cell B2 in `hatsize.wks` to also appear in cell C14 of `wardrobe.wks`.
5. So that FinnogaCalc can send the value to FinnogaCalc_2, FinnogaCalc_2 creates an object spec for cell C14 by calling a ToolTalk function. This object is identified by an objid.
6. FinnogaCalc_2 then gives this objid to FinnogaCalc (for example, through the clipboard).
7. FinnogaCalc remembers that its cell B2 should appear in the object identified by this objid and sends a message that contains the value.
8. ToolTalk routes the message. To deliver the message, the ToolTalk service:
    a. Examines the spec associated with the objid and finds that the type of the objid is `FinnogaCalc_cell` and that the corresponding object is in the file `wardrobe.wks`.
    b. Consults the otype definition for `FinnogaCalc_cell`. From the otype, the ToolTalk service determines that this message is observed by processes of ptype `FinnogaCalc` and that the scope of the message should be `TT_FILE`.
    c. Matches the message against registered patterns and locates all processes of this ptype that are observing the proper file. FinnogaCalc_2 matches, but FinnogaCalc does not.
    d. Delivers the message to FinnogaCalc_2.
9. FinnogaCalc_2 recognizes that the message contains an object that corresponds to cell C14. FinnogaCalc_2 updates the value in `wardrobe.wks` and displays the new value.

Otype Addressing

Sometimes you may need to send an object-oriented message without knowing the objid. To handle these cases, the ToolTalk service provides otype addressing. This addressing mode requires the sender to specify the operation, arguments, scope, and otype. The ToolTalk service looks in the specified otype definition for a message pattern that matches the message's operation and arguments to locate handling and observing processes. The dispatch and delivery then proceed as in messages to specific objects.

Modifying Applications to Send ToolTalk Messages

To send ToolTalk messages, your application must perform several operations: it must be able to create and complete ToolTalk messages; it must be able to add message callback routines; and it must be able to send the completed message.

Creating Messages

The ToolTalk service provides three methods to create and complete messages:

1. General-purpose function
   - `tt_message_create()`
2. Process-oriented notice and request functions
   - `tt_pnotice_create()`
   - `tt_prequest_create()`
3. Object-oriented notice and request functions
   - `tt_onotice_create()`
   - `tt_orequest_create()`

The process- and object-oriented notice and request functions make message creation simpler for the common cases. They are functionally identical to strings of other `tt_message_create()` and`tt_message_`<*attribute*>`_set()` calls, but are easier to write and read. Table 7-4 and Table 7-5 list the ToolTalk functions that are used to create and complete message

Table 7-4 Functions Used to Create Messages

ToolTalk Function	Description
`tt_onotice_create(const char *objid, const char *op)`	Creates an object-oriented notice.
`tt_orequest_create(const char *objid, const char *op)`	Creates an object-oriented request.
`tt_pnotice_create(Tt_scope scope, const char *op)`	Creates a process-oriented notice.
`tt_prequest_create(Tt_scope scope, const char *op)`	Creates a process-oriented request.
`tt_message_create(void)`	Creates a message. This function is the ToolTalk general purpose function to create messages.

Note - The return type for all the create functions is `Tt_message`.

Table 7-5 Functions Used to Complete Messages

ToolTalk Function	Description
`tt_message_address_set(Tt_message m, Tt_address p)`	Sets addressing mode (for example, point-to-point).
`tt_message_arg_add(Tt_message m, Tt_mode n, const char *vtype, const char *value)`	Adds a null-terminated string argument.
`tt_message_arg_bval_set(Tt_message m, int n, const unsigned char *value, int len)`	Sets an argument's value to the specified byte array.

Note - The return type for all the functions used to complete messages is `Tt_status`.

Table 7-5 Functions Used to Complete Messages *(Continued)*

ToolTalk Function	Description
`tt_message_arg_ival_set(Tt_message m, int n, int value)`	Sets an argument's value to the specified integer.
`tt_message_arg_val_set(Tt_message m, int n, const char *value)`	Sets an argument's value to the specified null-terminated string.
tt_message_barg_add(Tt_message m, Tt_mode n, const char *vtype, const unsigned char *value, int len)	Adds a byte array argument.
tt_message_iarg_add(Tt_message m, Tt_mode n, const char *vtype, int value)	Adds an integer argument.
tt_message_context_bval(Tt_message m, const char *slotname, unsigned char **value, int *len)	Gets a context's value to the specified byte array.
tt_message_context_ival(Tt_message m, const char *slotname, int *value)	Gets a context's value to the specified integer.
tt_message_context_val(Tt_message m, const char *slotname)	Gets a context's value to the specified string.
tt_message_icontext_set(Tt_message m, const char *slotname, int value)	Sets a context to the specified integer.
tt_message_bcontext_set(Tt_message m, const char *slotname, unsigned char *value, int length);	Sets a context to the specified byte array.
tt_message_context_set(Tt_message m, const char *slotname, const char *value)	Sets a context to the specified null-terminated string.
tt_message_class_set(Tt_message m, Tt_class c)	Sets the type of message (either notice or request).
tt_message_file_set(Tt_message m, const char *file)	Sets the file to which the message is scoped.
tt_message_handler_ptype_set(Tt_message m, const char *ptid)	Sets the ptype that is to receive the message.
tt_message_handler_set(Tt_message m, const char *procid)	Sets the procid that is to receive the message.
tt_message_object_set(Tt_message m, const char *objid)	Sets the object that is to receive the message.

Note - The return type for all the functions used to complete messages is `Tt_status`.

Table 7-5 Functions Used to Complete Messages *(Continued)*

ToolTalk Function	Description
tt_message_op_set(Tt_message m, const char *opname)	Sets the operation that is to receive the message.
tt_message_otype_set(Tt_message m, const char *otype)	Sets the object type that is to receive the message.
tt_message_scope_set(Tt_message m, Tt_scope s)	Sets the recipients who are to receive the message (file, session, both).
tt_message_sender_ptype_set(Tt_message m, const char *ptid)	Sets the ptype of the application that is sending the message.
tt_message_session_set(Tt_message m, const char *sessid)	Sets the session to which the message is scoped.
tt_message_status_set(Tt_message m, int status)	Sets the status of the message; this status is seen by the receiving application.
tt_message_status_string_set(Tt_message m, const char *status_str)	Sets the text that describes the status of the message; this text is seen be the receiving application.
tt_message_user_set(Tt_message m, int key, void *v)	Sets a message that is internal to the sending application; this internal message is opaque data that is not seen by the receiving application.

Note - The return type for all the functions used to complete messages is `Tt_status`.

Using the General-Purpose Function to Create ToolTalk Messages

You can use the general-purpose function `tt_message_create()` to create and complete ToolTalk messages. If you create a process- or object-oriented message with `tt_message_create()`, use the `tt_message_`<*attribute*>`_set()` calls to set the attributes.

Class

- Use TT_REQUEST for messages that return values or status. You will be informed when the message is handled or queued, or when a process is started to handle the request.
- Use TT_NOTICE for messages that only notify other processes of events.

Address

- Use TT_PROCEDURE to send the message to any process that can perform this operation with these arguments. Fill in op and args attributes of this message.
- Use TT_OTYPE to send the message to this type of object that can perform this operation with these arguments. Fill in otype, op, and args attributes of the message.
- Use TT_HANDLER to send the message to a specific process. Specify the handler attribute value.

  Usually, one process makes a general request, picks the handler attribute from the reply, and directs further messages to that handler. If you specify the exact procid of the handler, the ToolTalk service will deliver the message directly — no pattern matching is done and no other applications can observe the message. This point-to-point (PTP) message passing feature enables two processes to rendezvous through broadcast message passing and then communicate explicitly with one another.
- Use TT_OBJECT to send the message to a specific object that performs this operation with these arguments. Fill in *object*, *op*, and *args* attributes of this message.

Scope

Fill in the scope of the message delivery. Potential recipients could be joined to:

- TT_SESSION
- TT_FILE
- TT_BOTH
- TT_FILE_IN_SESSION

Depending on the scope, the ToolTalk service will add the default session or file, or both to the message.

Op

Fill in the operation that describes the notification or request that you are making. To determine the operation name, consult the ptype definition for the target recipient or the message protocol definition.

Args

Fill in any arguments specific to the operation. Use the function that best suits your argument's data type:

- `tt_message_arg_add()`

  Adds an argument whose value is a zero-terminated character string.

- `tt_message_barg_add()`

  Adds an argument whose value is a byte string.

- `tt_message_iarg_add()`

  Adds an argument whose value is an integer.

For each argument you add (regardless of the value type), specify:

- `Tt_mode`
  Specify `TT_IN` or `TT_INOUT`. `TT_IN` indicates that the argument is written by the sender and can be read by the handler and any observers. `TT_INOUT` indicates that the argument is written by the sender and the handler and can be read by all. If you are sending a request that requires the handler to provide an argument in return, use `TT_INOUT`.
- Value Type
  The value type (*vtype*) describes the type of argument data that is to be added. The ToolTalk service uses the vtype name when it compares a message to registered patterns to determine a message's recipients. The ToolTalk service does not use the vtype to process a message or pattern argument value.

  The vtype name helps the message receiver interpret data. For example, if a word processor rendered a paragraph into a PostScript representation in memory, it could call `tt_message_arg_add` with the following arguments:

  ```
  tt_message_arg_add (m, "PostScript", buf);
  ```

  In this case, the ToolTalk service would assume `buf` pointed to a zero-terminated string and send it.

Similarly, an application could send an enum value in a ToolTalk message; for example, an element of `Tt_status`:

```
tt_message_iarg_add(m, "Tt_status", (int) TT_OK);
```

The ToolTalk service sends the value as an integer but the "`Tt_status`" vtype tells the recipient what the value means.

Note – It is very important that senders and receivers define particular vtype names so that a receiver does not attempt to retrieve a value that was stored in another fashion; for example, a value stored as an integer but retrieved as a string.

Creating Process-Oriented Messages

You can easily create process-oriented notices and requests. To get a handle or opaque pointer to a new message object for a procedural notice or request, use the `tt_pnotice_create` or `tt_prequest_create` function. You can then use this handle on succeeding calls to reference the message.

When you create a message with `tt_pnotice_create` or `tt_prequest_create`, you must supply the following two attributes as arguments:

1. Scope

   Fill in the scope of the message delivery. Potential recipients could be joined to:

   - `TT_SESSION`
   - `TT_FILE`
   - `TT_BOTH`
   - `TT_FILE_IN_SESSION`

   Depending on the scope, the ToolTalk service fills in the default session or file (or both).

2. Op

   Fill in the operation that describes the notice or request you are making. To determine the operation name, consult the ptype definition for the target process or other protocol definition.

You use the `tt_message_`<*attribute*>`_set` calls to complete other message attributes such as operation arguments.

Creating and Completing Object-Oriented Messages

You can easily create object-oriented notices and requests. To get a handle or opaque pointer to a new message object for a object-oriented notice or request, use the `tt_onotice_create` or `tt_orequest_create` function. You can then use this handle on succeeding calls to reference the message.

When you create a message with `tt_onotice_create` or `tt_orequest_create`, you must supply the following two attributes as arguments:

1. Objid

   Fill in the unique object identifier.

2. Op

   Fill in the operation that describes the notice or request you are making. To determine the operation name, consult the ptype definition for the target process or other protocol definition.

You use the `tt_message_`<*attribute*>`_set` calls to complete other message attributes such as operation arguments.

Adding Message Callbacks

When a request contains a message callback routine, the callback routine is automatically called when the reply is received to examine the results of the reply and take appropriate actions.

Note – Callbacks are called in reverse order of registration (for example, the most recently added callback is called first).

You use `tt_message_callback_add` to add the callback routine to your request. When the reply comes back and the reply message has been processed through the callback routine, the reply message must be destroyed before the callback function returns `TT_CALLBACK_PROCESSED`. To destroy the reply message, use `tt_message_destroy`, as illustrated in the following example:

```
Tt_callback_action
sample_msg_callback(Tt_message m, Tt_pattern p)
{
    ... process the reply msg ...

    tt_message_destroy(m);
    return TT_CALLBACK_PROCESSED;
}
```

The following code sample is a callback routine, `cntl_msg_callback`, that examines the state field of the reply and takes action if the state is started, handled, or failed.

```
/*
 * Default callback for all the ToolTalk messages we send.
 */

Tt_callback_action
cntl_msg_callback(m, p)
     Tt_message m;
     Tt_pattern p;
{
    int     mark;
    char    msg[255];
    char    *errstr;


    mark = tt_mark();
    switch (tt_message_state(m)) {
          case TT_STARTED:
                         /*Put in code to start the editor*/
            break;
          case TT_HANDLED:
```

```
                /*Put in code to handle the message*/
                break;
            case TT_FAILED:
                errstr = tt_message_status_string(m);
                if (tt_pointer_error(errstr) == TT_OK && errstr) {
                sprintf(msg,"%s failed: %s", tt_message_op(m), errstr);
                } else if (tt_message_status(m) == TT_ERR_NO_MATCH) {
                sprintf(msg,"%s failed: Couldn't contact editor",
                    tt_message_op(m),
                    tt_status_message(tt_message_status(m)));
                } else {
                sprintf(msg,"%s failed: %s",
                    tt_message_op(m),
                    tt_status_message(tt_message_status(m)));
                }
                 /*Put in code to display the error message*/
                break;
            default:
                break;
        }
        /*
         * no further action required for this message. Destroy it
         * and return TT_CALLBACK_PROCESSED so no other callbacks will
         * be run for the message.
         */
        tt_message_destroy(m);
        tt_release(mark);
        return TT_CALLBACK_PROCESSED;
}
```

You can also add callbacks to static patterns by attaching a callback to the opnum of a signature in a ptype. When a message is delivered because it matched a static pattern with an opnum, the ToolTalk service checks for any callbacks attached to the opnum and runs them.

- Use `tt_otype_opnum_callback_add` to attach the callback routine to the opnum of an osignature.
- Use `tt_ptype_opnum_callback_add` to attach the callback routine to the opnum of a psignature.

Sending a Message

When you have completed your message, use `tt_message_send` to send it.

If the ToolTalk service returns `TT_WRN_STALE_OBJID`, it has found a forwarding pointer in the ToolTalk database that indicates the object mentioned in the message has been moved; however, the ToolTalk service will send the message with the new objid. You can then use `tt_message_object` to retrieve the new objid from the message and put it into your internal data structure.

If you will not need the message in the future (for example, if the message was a notice), you can use `tt_message_destroy` to delete the message and free storage space.

Note – If you are expecting a reply to the message, do not destroy the message until you have handled the reply.

Examples

This example illustrates how to create and send a pnotice:

```
/*
* Create and send a ToolTalk notice message
* ttsample1_value(in int <new value)
*/

msg_out = tt_pnotice_create(TT_SESSION, "ttsample1_value");
tt_message_arg_add(msg_out, TT_IN, "integer", NULL);
```

```
    tt_message_arg_ival_set(msg_out, 0, 42;
    tt_message_send(msg_out);

    /*
    * Since this message is a notice, we don't expect a reply, so
    * there's no reason to keep a handle for the message.
    */

    tt_message_destroy(msg_out);
```

Here is an illustration of how an orequest is created and sent when the callback routine for `cntl_ui_hilite_button` is called:

```
/*
 * Notify callback function for 'cntl_ui_hilite_button'.
 */
void
cntl_ui_hilite_button_handler(item, event)
    Panel_itemitem;
    Event   *event;
{
    Tt_messagemsg;

    if (cntl_objid == (char *)0) {
        /*Put in code to display an error*/
        return;
    }
    msg = tt_orequest_create(cntl_objid, "hilite_obj");
    tt_message_arg_add(msg, TT_IN, "string", cntl_objid);
    tt_message_callback_add(msg, cntl_msg_callback);
    tt_message_send(msg);
}
```

Message Patterns 8

This chapter describes how to provide message pattern information to the ToolTalk service. The ToolTalk service uses message patterns to determine message recipients. After receiving a message, the ToolTalk service compares the message to all current message patterns to find a matching pattern. Once a match is made, the message is delivered to the application that registered the message pattern.

You can provide message pattern information to the ToolTalk service using either dynamic or static methods, or both. The method you choose depends on the type of messages you want to receive.

- If the types of messages you want to receive will vary while your application is running, the *dynamic method* allows you to add, change, or remove message pattern information after your application has started.
- If you want a message to start your application or to be queued if your application is not running, the *static method* provides an easy way to specify these instructions. The static method also provides an easy way to specify the message pattern information if you want to receive a defined set of messages. For more information, see Chapter 10, "Static Message Patterns".

Regardless of the method you choose to provide message patterns to the ToolTalk service, you will want to update these patterns with each current session and file information so that you receive all messages that reference the session or file in which you are interested.

Message Pattern Attributes

The attributes in your message pattern specify the type of messages you want to receive. Although some attributes are set and have only one value, you can supply multiple values for most of the attributes you add to a pattern.

Table 8-1 provides a complete list of attributes you can put in your message patterns.

Table 8-1 ToolTalk Message Pattern Attributes

Pattern Attribute	Value	Description
Category	`TT_OBSERVE` `TT_HANDLE`	Declares whether you want to perform the operation listed in a message or only observe a message.

Scope	`TT_SESSION`
	`TT_FILE`
	`TT_FILE_IN_SESSION`
	`TT_BOTH`
		Declares interest in messages about a session or a file, or both; join a session or file after the message pattern is registered to update the sessid and filename.

Arguments	`arguments or results`	Declares the positional arguments for the operation in which you are interested.
Context	`<name, value>`	Declares the keyword or non-positional arguments for the operation in which you are interested
Class	`TT_NOTICE` `TT_REQUEST`	Declares whether you want to receive notices or requests, or both.
File	`char *pathname`	Declares the files in which you are interested. If the scope of the pattern does not require a file, the file is an attribute only.
Object	`char *objid`	Declares what objects in which you are interested.
Operation	`char *opname`	Declares the operations in which you are interested.

Table 8-1 ToolTalk Message Pattern Attributes *(Continued)*

Pattern Attribute	Value	Description
Otype	`char *otype`	Declares the type of objects in which you are interested.

address	`TT_PROCEDURE`
	`TT_OBJECT`
	`TT_HANDLER`
	`TT_OTYPE`
		Declares the type of address in which you are interested.

disposition	`TT_DISCARD`
	`TT_QUEUE`
	`TT_START`
	`TT_START+TT_QUEUE`
		Instructs the ToolTalk service how to handle messages to your application if an instance is not currently running.

sender	`char *procid`	Declares the sender in which you are interested.
sender_ptype	`char *ptype`	Declares the type of sending process in which you are interested.
session	`char *sessid`	Declares the session in which you are interested.

state	`TT_CREATED`
	`TT_SENT`
	`TT_HANDLED`
	`TT_FAILED`
	`TT_QUEUED`
	`TT_STARTED`
	`TT_REJECTED`
		Declares the state of the message in which you are interested.

All your message patterns must at least specify:

- **Category** — Whether the application wants to perform operations listed in messages or only view messages.
  - Use `TT_OBSERVE` if you want to observe messages.
  - Use `TT_HANDLE` if you want to perform operations requested by the messages.
- **Scope** — Whether the application is interested in messages about a particular session or file.

- Use TT_SESSION to receive messages from other processes in your session.
- Use TT_FILE to receive messages about the file you have joined.
- Use TT_FILE_IN_SESSION to receive messages for the file you have joined while in this session.
- Use TT_BOTH to receive both messages for the file, the session, or the file and the session you have joined.

The ToolTalk service compares message attributes to pattern attributes as follows:

- The ToolTalk service counts the message attribute as matched if:
  - No pattern attribute is specified.
  - The pattern does not name a context slot.
  - The pattern has an empty context slot.

  The fewer pattern attributes you specify, the more messages you become eligible to receive.
- If there are multiple values specified for a pattern attribute, one of the values must match the message attribute value. If no value matches, the ToolTalk service will not consider your application as a receiver.
- If context slots are contained in the message, the ToolTalk service will not consider your application as a receiver unless:
  - A value specified in a context slot of a pattern matches the value specified in the message context slot.
  - When multiple context slots are specified in a message, each context slot value in the message matches a corresponding context slot value in the pattern.

Scope Attributes

You can specify the following types of scopes in your message patterns:

1. Scope to a session only.
2. Scope to a file only.
3. Scope only to a file in a particular session.
4. Scope to either or both a file and a session.

Note – File scopes are restricted to NFS® and UFS file systems; you cannot scope a file across other types of file systems (for example, a tmpfs file system).

Scoping Only to a Session

The type `TT_SESSION` scopes only to a session. Static session-scoped patterns require an explicit `tt_session_join` call to set the scope value; dynamic session-scoped patterns can be set with either the `tt_session_join` call or the `tt_pattern_session_add` call.

Note – The session specified by these calls must be the default session.

The following code example shows a static session-scoped pattern:

```
/* Obtain procid */
tt_open();

/* Ptype is scoped to session */
tt_ptype_declare(ptype);

/* Join session */
tt_session_join(tt_default_session());
```

Now here is an illustration of a dynamic session-scoped pattern:

```
/* Obtain procid */
tt_open();

/* Create pattern */
Tt_pattern pat = tt_create_pattern();

/* Add scope to pattern */
tt_pattern_scope_add(pat, TT_SESSION);

/* Add session to pattern */
tt_pattern_session_add (tt_default_session());

/* Register pattern */
tt_pattern_register(pat);
```

Scoping Only to a File

The type TT_FILE scopes only to a file. This code example shows a static file-scoped pattern: .

```
/* Obtain procid */
tt_open();

/* Ptype is scoped to file */
tt_ptype_declare(ptype);

/* Join file */
tt_file_join(file);
```

Now here is an example that shows a dynamic file-scoped pattern:

```
/* Obtain procid */
tt_open();

/* Create pattern */
Tt_pattern pat = tt_create_pattern();

/* Add scope to pattern */
tt_pattern_scope_add(pat, TT_FILE);

/* Add file to pattern */
tt_pattern_file_add (pat, file);

/* Register pattern */
tt_pattern_register(pat);
```

Scoping to a File in a Session

The type TT_FILE_IN_SESSION scopes to the specified file in the specified session only. A pattern with this scope set will only match messages that are scoped to both the file and the session. This code example adds the session and then registers the pattern.

```
/* Obtain procid */
tt_open();

/* Create pattern */
Tt_pattern pat = tt_create_pattern();

/* Add scope to pattern */
tt_pattern_scope_add(pat, TT_FILE_IN_SESSION);

/* Add file to pattern */
tt_pattern_file_add(pat, file);

/* Add session to pattern */
tt_pattern_session_add(pat, tt_default_session());

/* Register pattern */
tt_pattern_register(pat);
```

The following example registers the pattern and then joins a session:

```
/* Obtain procid */
tt_open();

/* Create pattern */
Tt_pattern pat = tt_create_pattern();

/* Add scope to pattern */
tt_pattern_scope_add(pat, TT_FILE_IN_SESSION);

/* Add file to pattern */
tt_pattern_file_add(pat, file);

/* Register pattern */
tt_pattern_register(pat);

/* Join session */
tt_session_join(tt_default_session());
```

This example sets the scope value for a static pattern:

```
/* Obtain procid */
tt_open();
```

```
/* Declare Ptype */
Tt_ptype_declare(ptype);

/* Join File */
tt_file_join(file);

/* Join session */
tt_session_join(tt_default_session());
```

Scoping to a File and/or a Session

A `TT_BOTH`-scoped pattern will match messages that are scoped to the file, the session, or the file and the session. When you use this scope, however, you must explicitly make a `tt_file_join` call; otherwise, the ToolTalk service will only match messages that are scoped to both the file and session of the registered pattern. The following show code examples of how to use this scope:

```
/* Obtain procid */
tt_open();

/* Create pattern */
Tt_pattern pat = tt_create_pattern();

/* Add scope to pattern */
tt_pattern_scope_add(pat, TT_BOTH);

/* Add session to pattern */
tt_pattern_session_add(pat, tt_default_session());

/* Add file to pattern */
tt_pattern_file_add (pat, file);

/* Register pattern */
tt_pattern_register(pat);
```

Or,

```
/* Obtain procid */
tt_open();

/* Declare Ptype */
Tt_ptype_declare(ptype);
```

```
/* Join file */
tt_file_join(file);

/* Join session */
tt_session_join(tt_default_session());
```

Adding Files to Scoped Patterns

To match `TT_SESSION`-scoped messages and `TT_SESSION`-scoped patterns that have the same file attributes, you can add file attributes to `TT_SESSION`-scoped patterns with the `tt_pattern_file_add` call, as shown in the following example:

Note – The file attribute values do not affect the scope of the pattern.

```
/* Obtain procid */
tt_open();

/* Create pattern */
Tt_pattern pat = tt_create_pattern();

/* Add scope to pattern */
tt_pattern_scope_add(pat, TT_SESSION);

/* Add session to pattern */
tt_pattern_session_add(tt_default_session());

/* Add first file attribute to pattern */
tt_pattern_file_add(pat, file1);

/* Add second file attribute to pattern */
tt_pattern_file_add(pat, file2);

/* Register pattern */
tt_pattern_register(pat);
```

Context Attributes

ToolTalk *contexts* are sets of <name, value> pairs explicitly included in both messages and patterns. ToolTalk contexts allow fine-grain matching.

You can use contexts to associate arbitrary pairs with ToolTalk messages and patterns, and to restrict the set of possible recipients of a message. One common use of the restricted pattern matching provided by ToolTalk context attributes is to create sub-sessions. For example, two different programs could be debugged simultaneously with tools such as a browser, an editor, a debugger, and a configuration manager active for each program. The message and pattern context slots for each set of tools contain different values; the normal ToolTalk pattern matching of these values keeps the two sub-sessions separate.

Another use for the restricted pattern matching provided by ToolTalk context attributes is to provide information in environment variables and command line arguments to tools started by the ToolTalk service.

Disposition Attributes

Disposition attributes instruct the ToolTalk service how to handle messages to your application if an instance of the application is not currently running.

The disposition value specified in the static type definition of a pattern is the default disposition; however, if the message disposition specifies the handler ptype, the default disposition value is over-ridden. For example, a message disposition specifies a static type definition for the ptype *UWriteIt,* which includes the message signature *Display*. This message signature does not match any of the static signatures in the pattern. The ToolTalk service will follow the instructions for the disposition set in the message; for example, if the message disposition is `TT_START` and the *UWriteIt* ptype specifies a start string, the ToolTalk service will start an instance of the application if one is not running.

Dynamic Message Patterns 9

Defining Dynamic Messages

The dynamic method provides message pattern information while your application is running. You create a message pattern and register it with the ToolTalk service. You can add callback routines to dynamic message patterns that the ToolTalk service will call when it matches a message to the pattern.

To create and register a dynamic message pattern, you allocate a new pattern object, fill in the proper information, and register it. When you are done with the pattern (that is, when you are no longer interested in messages that match it), either unregister or destroy the pattern. You can register and unregister dynamic message patterns as needed.

The ToolTalk functions used to create, register, and unregister dynamic message patterns are listed in Table 9-1.

Table 9-1 Functions for Creating, Updating, and Deleting Message Patterns

ToolTalk Function	Description
`tt_pattern_create(void)`	Create Pattern
`tt_pattern_arg_add(Tt_pattern p, Tt_mode n, const char *vtype, const char *value)`	Add string arguments
`tt_pattern_barg_add(Tt_pattern m, Tt_mode n, const char *vtype, const unsigned char *value, int len)`	Add byte array arguments
`tt_pattern_iarg_add(Tt_pattern m, Tt_mode n, const char *vtype, int value)`	Add integer arguments
`tt_pattern_xarg_add(Tt_pattern m, Tt_mode n, const char *vtype, xdrproc_t xdr_proc, void *value)`	Adds an xdr argument to a byte array
`tt_pattern_bcontext_add(Tt_pattern p, const char *slotname, const unsigned char *value, int length)`	Add byte array contexts
`tt_pattern_context_add(Tt_pattern p, const char *slotname, const char *value)`	Add string contexts
`tt_pattern_icontext_add(Tt_pattern p, const char *slotname, int value)`	Add integer contexts
`tt_pattern_address_add(Tt_pattern p, Tt_address d)`	Add address
`tt_pattern_callback_add(Tt_pattern p, Tt__message_callback_action f)`	Add message callback
`tt_pattern_category_set(Tt_pattern p, Tt_category c)`	Set category
`tt_pattern_class_add(Tt_pattern p, Tt_class c)`	Add class
`tt_pattern_disposition_add(Tt_pattern p, Tt_disposition r)`	Add disposition

Note - The return type for all functions except `tt_pattern_create` is `Tt_status`; `tt_pattern_create` returns `Tt_pattern`.

Table 9-1 Functions for Creating, Updating, and Deleting Message Patterns *(Continued)*

ToolTalk Function	Description
`tt_pattern_file_add(Tt_pattern p, const char *file)`	Add file
`tt_pattern_object_add(Tt_pattern p, const char *objid)`	Add object
`tt_pattern_op_add(Tt_pattern p, const char *opname)`	Add operation
`tt_pattern_opnum_add(Tt_pattern p, int opnum)`	Add operation number
`tt_pattern_otype_add(Tt_pattern p, const char *otype)`	Add object type
`tt_pattern_scope_add(Tt_pattern p, Tt_scope s)`	Ad scope
`tt_pattern_sender_add(Tt_pattern p, const char *procid)`	Add sending process identifier
`tt_pattern_sender_ptype_add(Tt_pattern p, const char *ptid)`	Add sending process type
`tt_pattern_session_add(Tt_pattern p, const char *sessid)`	Add session identifier
`tt_pattern_state_add(Tt_pattern p, Tt_state s)`	Add state
`tt_pattern_user_set(Tt_pattern p, int key, void *v)`	Set user
`tt_pattern_register(Tt_pattern p)`	Register pattern
`tt_pattern_unregister(Tt_pattern p)`	Unregister pattern
`tt_pattern_destroy(Tt_pattern p)`	Destroy message pattern

Note - The return type for all functions except `tt_pattern_create` is `Tt_status`; `tt_pattern_create` returns `Tt_pattern`.

Creating a Message Pattern

To create message patterns, use the tt_pattern_create function. You can use this function to get a handle or opaque pointer to a new pattern object, and then use this handle on succeeding calls to reference the pattern.

To fill in pattern information, use the tt_pattern_<*attribute*>_add and tt_pattern_<*attribute*>_set calls. You can supply multiple values for each attribute you add to a pattern. The pattern attribute matches a message attribute if any of the values in the pattern match the value in the message. If no value is specified for an attribute, the ToolTalk service assumes that you want any value to match. Some attributes are set and, therefore, can only have one value.

Adding a Message Pattern Callback

To add a callback routine to your pattern, use the tt_pattern_callback_add function.

Note – Callbacks are called in reverse order of registration (for example, the most recently added callback is called first).

When the ToolTalk service matches a message, it automatically calls your callback routine to examine the message and take appropriate actions. When a message that matches a pattern with a callback is delivered to you, it is processed through the callback routine. When the routine is finished, it returns TT_CALLBACK_PROCESSED and the API objects involved in the operation are freed. You can then use tt_message_destroy to destroy the message, which frees the storage used by the message, as illustrated in the following code sample:

```
Tt_callback_action
sample_msg_callback(Tt_message m, Tt_pattern p)
{
    ... process the reply msg ...

    tt_message_destroy(m);
    return TT_CALLBACK_PROCESSED;
}
```

Registering a Message Pattern

To register the completed pattern, use the `tt_pattern_register` function. After you register your pattern, join the sessions or files of interest.

The following code sample creates and registers a pattern.

```
/*
 * Create and register a pattern so ToolTalk knows we are
 * interested in "ttsample1_value" messages within the
 * session we join.
 */

pat = tt_pattern_create();
tt_pattern_category_set(pat, TT_OBSERVE);
tt_pattern_scope_add(pat, TT_SESSION);
tt_pattern_op_add(pat, "ttsample1_value");
tt_pattern_register(pat);
```

Deleting and Unregistering a Message Pattern

Note – If delivered messages that matched the deleted pattern have not been retrieved by your application (for example, the messages might be queued), the ToolTalk service does not destroy these messages.

To delete a message pattern, use the `tt_pattern_destroy` function. This function first unregisters the pattern and then destroys the pattern object.

To stop receiving messages that match a message pattern without destroying the pattern object, use the `tt_pattern_unregister` to unregister the pattern.

The ToolTalk service will automatically unregister and destroy all message pattern objects when you call `tt_close`.

Updating Message Patterns with the Current Session or File

To update your message patterns with the session or file in which you are currently interested, join the session or file.

Joining the Default Session

When you join a session, the ToolTalk service updates your message pattern with the sessid. For example, if you have declared a ptype or registered a message pattern that specifies `TT_SESSION` or `TT_FILE_IN_SESSION`, use `tt_session_join` to join the default session. The following code sample shows how to join the default session.

```
/*
 * Join the default session
 */

tt_session_join(tt_default_session());
```

Table 9-2 lists the ToolTalk functions you use to join the session in which you are interested.

Table 9-2 ToolTalk Functions for Joining Default Sessions

Return Type	ToolTalk Function	Description
`char *`	`tt_default_session(void)`	Return default session id.
`Tt_status`	`tt_default_session_set(const char *sessid)`	Set default session.
`char *`	`tt_initial_session(void)`	Return initial session id.
`Tt_status`	`tt_session_join(const char *sessid)`	Join this session .
`Tt_status`	`tt_session_quit(const char *sessid)`	Quit session.

Once your patterns are updated, you will begin to receive messages scoped to the session you joined.

Note – If you had previously joined a session and then registered a ptype or a new message pattern, you must again join the same session or a new session to update your pattern before you will receive messages that match your new pattern.

When you no longer want to receive messages that reference the default session, use the `tt_session_quit` function. This function removes the sessid from your session-scoped message patterns.

Joining Multiple Sessions

When you join multiple sessions, you will automatically get responses to requests and point-to-point messages, but you will not get notices unless you explicitly join the new session. The following code sample shows how to join the multiple sessions.

```
tt_default_session_set(new_session_identifier);
tt_open();
tt_session_join(new_session);
```

In order to effectively use multiple sessions, you must store the session ids of the sessions in which you are interested in order to pass these identifiers to `tt_default_session_set` prior to opening a new session with `tt_open`; that is, you need to place the values (which `ttsession` stores in the environment variable `TT_SESSION`) in a file on the system so that other ToolTalk clients can access the value of a session id contained in that file and use it to open the non-default session. For example, you can store the session ids in a "well-known" file and then send a file-scoped message (indicating this file) to all clients that have registered an appropriate pattern. The client will then know to open the scoped-to file, read one or more session ids from it, and use these session ids (with `tt_open`) to open a non-default session. An alternative method is advertising the session ids by means of, for example, a name service or a third-party database.

Note – How `ttsession` session ids are stored and passed to interested clients is beyond the scope of the ToolTalk protocol and must be determined based on the architecture of the system.

Joining Files of Interest

When you join a file, the ToolTalk service automatically adds the name of the file to your file-scoped message patterns. For example, if you have declared a process type or registered a message pattern that specifies `TT_FILE` or `TT_FILE_IN_SESSION`, use the `tt_file_join` function to join files of interest. Table 9-3 lists the ToolTalk functions you use to express your interest in specific files.

Table 9-3 ToolTalk Functions for Joining Files of Interest

Return Type	ToolTalk Function	Description
`char *`	`tt_default_file(void)`	Join default file.
`Tt_status`	`tt_default_file_set(const char *docid)`	Set default file.
`Tt_status`	`tt_file_join(const char *filepath)`	Join this file.
`Tt_status`	`tt_file_quit(const char *filepath)`	Quit file.

When you no longer want to receive messages that reference the file, use the `tt_file_quit` function to remove the file name from your file-scoped message patterns.

Static Message Patterns 10

Defining Static Messages

The static messaging method provides an easy way to specify the message pattern information if you want to receive a defined set of messages.

To use the static method, you define your process types and object types and compile them with the ToolTalk type compiler, `tt_type_comp`. When you declare your process type, the ToolTalk service creates message patterns based on that type. These static message patterns remain in effect until you close communication with the ToolTalk service.

Defining Process Types

Your application can still be considered a potential message receiver even when no process is running the application. To do this, you provide message patterns and instructions on how to start the application in a process type (*ptype*) file. These instructions tell the ToolTalk service to perform one of the following actions when a message is available for an application but the application is not running:

- Start the application and deliver the message.
- Queue the message until the application is running.
- Discard the message.

To make the information available to the ToolTalk service, the ptype file is compiled with the ToolTalk type compiler, `tt_type_comp`, at application installation time.

When an application registers a ptype with the ToolTalk service, the message patterns listed in it are automatically registered, too.

Ptypes provide application information that the ToolTalk service can use when the application is not running. This information is used to start your process, if necessary, to receive a message or queue messages until the process starts.

A ptype begins with a process-type identifier (*ptid*). Following the ptid are:

1. An optional start string — The ToolTalk service will execute this command, if necessary, to start a process running the program.
2. 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.

For an example of a ptypes file, see the following file:
`/usr/dt/examples/tt/edit_demo/edit.types.model`.

Signatures

Signatures describe the messages that the program wants to receive. A signature is divided by an arrow (=>) into two parts. The first part of a signature specifies matching attribute values. The more attribute values specified in a signature, the fewer messages the signature will match. The second part of a signature specifies receiver values that the ToolTalk service will copy into messages that match the first part of the signature.

A ptype signature can contain values for disposition and operation numbers (*opnum*). The ToolTalk service uses the disposition value (start, queue, or the default discard) to determine what to do with a message that matches the signature when no process is running the program. The opnum value is provided as a convenience to message receivers. When two signatures have the same operation name but different arguments, different opnums make incoming messages easy to identify.

Creating a Ptype File

The following shows the syntax for a ptype file:

```
ptype::='ptype' ptid '{'
        property*
        ['observe:' psignature*]
        ['handle:' psignature* ]
        '}' [';']
property::=property_id value ';'
property_id::='start'
value::=string
ptid::= identifier
psignature::=[scope] op args [contextdcl]
        ['=>'
        ['start']['queue']
        ['opnum='number]]
        ';'
scope::='file'
    |   'session'
    |   'file_in_session'
args::= '(' argspec {, argspec}* ')'
    |   '(void)'
    |   '()'
contextdcl::='context' '(' identifier {, identifier}* ')' ';'
argspec::=mode type name
mode::= 'in' | 'out' | 'inout'
type::= identifier
name::= identifier
```

Property_id Information

ptid

process-type identifier (ptid). Identifies the process type. A ptid must be unique for every installation. Because this identifier cannot be changed after installation time, each chosen name must be unique. For example, you can use a name that includes the trademarked name of your product or company. The ptid cannot exceed 32 characters and should not be one of the reserved identifiers: ptype, otype, start, opnum, queue, file, session, observe, or handle.

start

Start string for the process. If the ToolTalk service needs to start a process, it executes this command; `/usr/bin/sh` is used as the shell.

Before executing the command, the ToolTalk service defines `TT_FILE` as an environment variable with the value of the file attribute of the message that started the application. This command runs in the environment of `ttsession`, not in the environment of the sender of the message that started the application, so any context information must be carried by message arguments or contexts.

Psignature Matching Information

scope

This pattern attribute is matched against the scope attribute in messages.

op

Operation name. This name is matched against the op attribute in messages.

Note – If you specify message signatures in both your ptype and otypes, use unique operation names in each. For example, do not specify a display operation in both your ptype and otype.

---

args

Arguments for the operation. If the args list is `(void)`, the signature matches only messages with no arguments. If the args list is empty (that is, "`()`"), the signature matches without regard to the arguments.

contextdcl

Context name. When a pattern with this named context is generated from the signature, it contains an empty value list.

Psignature Actions Information

start

If the psignature matches a message and no running process of this ptype has a pattern that matches the message, start a process of this ptype.

queue

If the psignature matches a message and no running process of this ptype has a pattern that matches the message, queue the message until a process of this ptype registers a pattern that matches it.

opnum

Fill in the message's opnum attribute with the specified number to enable you to identify the signature that matched the message.

When the message matches the signature, the opnum from the signature is filled into the message. Your application can then retrieve the opnum with the `tt_message_opnum` call. By giving each signature a unique opnum, you can quickly determine which signature matched the message.

You can attach a callback routine to the opnum with the `tt_ptype_opnum_callback_add` call. When the message is matched, the ToolTalk service will check for any callbacks attached to the opnum and, if any are found, run them.

The `edit.types.model` ptypes file and the source file, `edit.c` (located in the `/usr/dt/examples/tt/edit_demo` directory), both include `CDE_EditDemo_opnums.h`. This allows both files to share the same definitions.

Defining Object Types

When a message is addressed to a specific object or a type of object, the ToolTalk service must be able to determine to which application the message is to be delivered. Applications provide this information in an *object type* (*otype*). An otype names the ptype of the application that manages the object and describes message patterns that pertain to the object.

These message patterns also contain instructions that tell the ToolTalk service what to do if a message is available but the application is not running. In this case, ToolTalk performs one of the following instructions:

- Start the application and deliver the message.
- Queue the message until the application is running.
- Discard the message.

To make the information available to the ToolTalk service, the otype file is compiled with the ToolTalk type compiler `tt_type_comp` at application installation time. When an application that manages objects registers with the ToolTalk service, it declares its ptype. When a ptype is registered, the ToolTalk service checks for otypes that mention the ptype and registers the patterns found in these otypes.

The otype for your application provides addressing information that the ToolTalk service uses when delivering object-oriented messages. The number of otypes you have, and what they represent, depends on the nature of your application. For example, a word processing application might have otypes for characters, words, paragraphs, and documents; a diagram editing application might have otypes for nodes, arcs, annotation boxes, and diagrams.

An otype begins with an object-type identifier (*otid*). Following the otid are:

1. An optional start string — ToolTalk will execute this command, if necessary, to start a process running the program.
2. Signatures — Code that defines the messages that can be addressed to objects of the type (that is, the operations that can be invoked on objects of the type).

Signatures

Signatures defines the messages that can be addressed to objects of the type. A signature is divided by an arrow (=>) into two parts. The first part of a signature defines matching criteria for incoming messages. The second part of a signature defines receiver values that the ToolTalk service adds to each message that matches the first part of the signature. These values specify the ptid of the program that implements the operation and the message's scope and disposition.

Creating Otype Files

The following shows the syntax for an otype file:

```
otype::=obj_header'{' objbody* '}' [';']
obj_header::='otype' otid [':' otid+]
objbody::='observe:' osignature*
    |   'handle:' osignature*

osignature::=op args [contextdcl] [rhs][inherit] ';'
rhs ::= ['=>' ptid [scope]]
        ['start']['queue']
        ['opnum='number]
inherit::='from' otid
args::= '(' argspec {, argspec}* ')'
    |   '(void)'
    |   '()'
contextdcl::='context' '(' identifier {, identifier}* ')' ';'
argspec::=mode type name
mode::= 'in' | 'out' | 'inout'
type::= identifier
name::= identifier
otid::= identifier
ptid::= identifier
```

Obj_Header Information

otid

object type identifier (otid). Identifies the object type. An otid must be unique for every installation. Because this identifier cannot be changed after installation time, each chosen name must be unique. For example, begin with the ptid of the tool that implements the otype. The otid is limited to 64 characters and should not be one of the reserved identifiers: ptype, otype, start, opnum, start, queue, file, session, observe, or handle.

Osignature Information

The object body portion of the otype definition is a list of osignatures for messages about the object that your application wants to observe and handle.

op

Operation name. This name is matched against the op attribute in messages.

Note – If you specify message signatures in both your ptype and otypes, use unique operation names in each. For example, do not specify a display operation in both your ptype and otype.

---

args

Arguments for the operation. If the args list is `(void)`, the signature matches only messages with no arguments. If the args list is empty (just "()"), the signature matches messages without regard to the arguments.

contextdcl

Context name. When a pattern with this named context is generated from the signature, it contains an empty value list.

ptid

Process type identifier for the application that manages this type of object.

opnum

Fill in the message's opnum attribute with the specified number to enable you to identify the signature that matched the message.

When the message matches the signature, the opnum from the signature is filled into the message. Your application can then retrieve the opnum with the `tt_message_opnum` call. By giving each signature a unique opnum, you can quickly determine which signature matched the message.

You can attach a callback routine to the opnum with the `tt_otype_opnum_callback_add` call. When the message is matched, the ToolTalk service will check for any callbacks attached to the opnum and, if any are found, run them.

inherit

Otypes form an inheritance hierarchy in which operations can be inherited from base types. The ToolTalk service requires the otype definer to explicitly name all inherited operations and the otype from which to inherit. This explicit naming prevents later changes (such as adding a new level to the hierarchy, or adding new operations to base types) from unexpectedly affecting the behavior of an otype.

scope

This pattern attribute is matched against the scope attribute in messages. It appears on the rightmost side of the arrow and is filled in by the ToolTalk service during message dispatch. This means the definer of the otype can specify the attributes instead of requiring the message sender to know how the message should be delivered.

Osignature Actions Information

start

If the osignature matches a message and no running process of this otype has a pattern that matches the message, start a process of this otype.

queue

If the osignature matches a message and no running process of this otype has a pattern that matches the message, queue the message until a process of this otype registers a pattern that matches it.

See the `edit.types.model` file for an example otype definition.

Installing Type Information

The ToolTalk Types Database makes ptype and otype information available on the host that executes the sending process, the host that executes the receiving process, and the hosts that run the sessions to which the processes are joined.

- To start applications and to queue messages, the ptype definition must be placed into the ToolTalk Types Database.
- To receive messages addressed to objects your application creates and manages, the otype definitions must also be installed in the ToolTalk Types Database.

To place your type information into the ToolTalk Types Database and make it available to the ToolTalk service, you compile your type files with the ToolTalk type compiler, `tt_type_comp`. This compiler creates ToolTalk types definitions for your type information and stores them in the ToolTalk Types Database. See Chapter 4, "Maintaining Application Information" for detailed information.

This version of the ToolTalk service provides a function to merge a compiled ToolTalk type file into the 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.

Checking for Existing Process Types

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

```
tt_ptype_exists(const char *ptid)
```

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

Ptypes

Table 10-1 Ptypes

Return Type	ToolTalk Function	Description
`Tt_status`	`tt_ptype_declare(const char *ptid)`	Registers the process type with the ToolTalk service.
`Tt_status`	`tt_ptype_exists(const char *ptid)`	Indicates whether a ptype is already installed.
`Tt_status`	`tt_ptype_undeclare(const char *ptid)`	Undeclares a ptype.

Declaring Process Type

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

To declare your ptype, use `tt_ptype_declare` during your application's ToolTalk initialization routine. The ToolTalk service will create the message patterns listed in your ptype and any otypes that reference the specified ptype.

The message patterns created when you declare your ptype exist in memory until your application exits the ToolTalk session.

Note – The message patterns created when you declare your ptype information cannot be unregistered with `tt_pattern_unregister`; however, you can unregister these patterns with `tt_ptype_undeclare`.

The example below illustrates how a ptype is registered during a program's initialization.

```
/*
 * Initialize our ToolTalk environment.
 */
int
edit_init_tt()
{
        int     mark;
        char    *procid = tt_open();
        int     ttfd;
        void    edit_receive_tt_message();
```

```
        mark = tt_mark();

        if (tt_pointer_error(procid) != TT_OK) {
                return 0;
        }
        if (tt_ptype_declare("CDE_EditDemo") != TT_OK) {
                fprintf(stderr,"CDE_EditDemo is not an installed
ptype.\n");
                return 0;
        }
        ttfd = tt_fd();
        tt_session_join(tt_default_session());
        notify_set_input_func(edit_ui_base_window,
                              (Notify_func)edit_receive_tt_message,
                              ttfd);

        /*
         * Note that without tt_mark() and tt_release(), the above
         * combination would leak storage -- tt_default_session()
returns
        * a copy owned by the application, but since we don't assign
the
         * pointer to a variable we could not free it explicitly.
         */

        tt_release(mark);
        return 1;
}
```

Undeclaring Process Types

There may be cases when you need to retract a declared ptype:

- An installation sets up a compile server that declares itself willing to accept compilation requests when it comes up. Once the server has accepted a request, it changes state and will no longer accept new compilation requests.
- A generic encapsulation process declares itself as multiple ptypes and then forwards requests to underlying tools. If an underlying tool exits, the generic wrapper no longer wants to declare itself as the ptype associated with that tool.

To unregister a ptype, use `tt_ptype_undeclare`. This call reverses the effect of the `tt_ptype_declare` call; that is, all patterns generated from the ptype are unregistered and the process is removed from the session's list of active processes with this ptype. This call returns a status of `TT_ERR_PTYPE` if the named ptype was not declared by the calling process.

Caution – One invocation of `tt_type_undeclare` will *completely* unregister the ptype regardless of how many times the process has declared the ptype; that is, multiple declarations of the ptype are the same as declaring it once.

The following is an example of how to retract a a declared ptype:

```
/* Obtain procid */
tt_open();

/* Undeclare Ptype */
tt_ptype_undeclare(ptype);
```

Receiving Messages 11

This chapter describes how to retrieve a message delivered to your application and how to handle the message once you have examined it. It also shows you how to send replies to requests that you receive.

To retrieve and handle ToolTalk messages, your application must perform several operations: it must be able to retrieve ToolTalk messages; it must be able to examine messages; it must provide callback routines; it must be able to respond to requests; and it must be able to destroy the message when it is no longer needed.

See the following demo program for an example of sending and receiving messages: `/usr/dt/examples/tt/broadcast.c`

Retrieving Messages

When a message arrives for your process, the ToolTalk-supplied file descriptor becomes active. When notified of the active state of the file descriptor, your process must call `tt_message_receive` to get a handle for the incoming message.

Note – When a message-arrived callback is invoked, the callback function should call tt_message_receive to receive the message. If a client fails to receive messages, ttsession could hang and be unusable.

The following illustrates how to receive a message:

```
/*
 * When a ToolTalk message is available, receive it.
 */
void
receive_tt_message()
{
    Tt_message msg_in;

    msg_in = tt_message_receive();

    /*
     * It's possible that the file descriptor would become active
     * even though ToolTalk doesn't really have a message for us.
     * The returned message handle is NULL in this case.
     */

    if (msg_in == NULL) return;
```

Handles for messages remain constant. For example, when a process sends a message, both the message and any replies to the message have the same handle as the sent message. Here is an example of how you can check the message state for `TT_HANDLED`:

```
Tt_message m, n;
m = tt_message_create();
...
tt_message_send(m);

... wait around for tt_fd to become active

n = tt_message_receive();
```

Code Checking the Message State *(Continued)*

Identifying and Processing Messages Easily

To easily identify and process messages you receive:

- Add a callback to a dynamic pattern with `tt_pattern_callback_add`. When you retrieve the message, the ToolTalk service will invoke any message or pattern callbacks. See Chapter 9, “Dynamic Message Patterns“ for more information on placing callbacks on patterns.

- Retrieve the message's opnum if you are receiving messages that match your ptype message patterns.

Recognizing and Handling Replies Easily

To easily recognize and handle replies to messages you send:

- Place specific callbacks on requests before you send them with `tt_message_callback_add`. See Chapter 7, "Sending Messages" for more information on placing callbacks on messages.
- Compare the handle of the message you sent with the message you just received. The handles will be the same if the message is a reply.
- Add information meaningful to your application on the request with the `tt_message_user_set` call.

Checking Message Status

When you receive a message, you must check its status. If the status is `TT_WRN_START_MESSAGE`, you must either reply to, reject, or fail the message (even if the message is a notice), or issue a `tt_message_accept` call.

Examining Messages

When your process receives a message, you examine the message and take appropriate action.

Before you start to retrieve values, obtain a mark on the ToolTalk API stack so that you can release the information the ToolTalk service returns to you all at once. The following example allocates storage, examines message contents, and releases the storage:

```
/*
 * Get a storage mark so we can easily free all the data
 * ToolTalk returns to us.
 */

mark = tt_mark();

tt_message_destroy(msg_in);
tt_release(mark);
return;
```

Table 11-1 lists the ToolTalk functions you use to examine the attributes of a message you have received.

Table 11-1 Functions to Examine Message Attributes

Return Type	ToolTalk Function	Description
`Tt_address`	`tt_message_address(Tt_message m)`	The address of the message.
`Tt_status`	`tt_message_arg_bval(Tt_message m, int n, unsigned char **value, int *len)`	The argument value as a byte array.
`Tt_status`	`tt_message_arg_ival(Tt_message m, int n, int *value)`	The argument value as an integer.
`Tt_status`	`tt_message_arg_xval(Tt_message m, int n, xdrproc_t xdr_proc, void *value)`	The argument value as an xdr.
`Tt_mode`	`tt_message_arg_mode(Tt_message m, int n)`	The argument mode (in, out, inout).
`char *`	`tt_message_arg_type(Tt_message m, int n)`	The argument type.
`char *`	`tt_message_arg_val(Tt_message m, int n)`	The argument value as a string.
`int`	`tt_message_args_count(Tt_message m)`	The number of arguments.
`Tt_class`	`tt_message_class(Tt_message m)`	The type of message (notice or request).
`int`	`tt_message_contexts_count(Tt_message m)`	The number of contexts.
`char *`	`tt_message_context_slotname(Tt_message m, int n)`	The name of a message's *nth* context.
`Tt_disposition`	`tt_message_disposition(Tt_message m)`	How to handle the message if there is no receiving application running.

Table 11-1 Functions to Examine Message Attributes *(Continued)*

Return Type	ToolTalk Function	Description
`char *`	`tt_message_file(Tt_message m)`	The name of the file to which the message is scoped.
`gid_t`	`tt_message_gid(Tt_message m)`	The group identifier of the sending application.
`char *`	`tt_message_handler(Tt_message m)`	The procid of the handler.
`char *`	`tt_message_handler_ptype(Tt_message m)`	The ptype of the handler.
`char *`	`tt_message_object(Tt_message m)`	The object to which the message was sent.
`char *`	`tt_message_op(Tt_message m)`	The operation name.
`int`	`tt_message_opnum(Tt_message m)`	The operation number.
`char *`	`tt_message_otype(Tt_message m)`	The object type to which the message was sent.
`Tt_pattern`	`tt_message_pattern(Tt_message m)`	The pattern to which the message is to be matched.
`Tt_scope`	`tt_message_scope(Tt_message m)`	Who is to receive the message (FILE, SESSION, BOTH)
`char *`	`tt_message_sender(Tt_message m)`	The procid of the sending application.
`char *`	`tt_message_sender_ptype(Tt_message m)`	The ptype of the sending application.
`char *`	`tt_message_session(Tt_message m)`	The session from which the message was sent.
`Tt_state`	`tt_message_state(Tt_message m)`	The current state of the message.
`int`	`tt_message_status(Tt_message m)`	The current status of the message.

Table 11-1 Functions to Examine Message Attributes *(Continued)*

Return Type	ToolTalk Function	Description
`char *`	`tt_message_status_string(Tt_message m)`	Text describing the current status of the message.
`uid_t`	`tt_message_uid(Tt_message m)`	The user identifier of the sending application.
`void *`	`tt_message_user(Tt_message m, int key)`	Opaque data internal to the application.

Callback Routines

You can tell the ToolTalk service to invoke a callback when a message arrives because a pattern has been matched.

```
p = tt_pattern_create();
tt_pattern_op_add(p, "EDIT");
... other pattern attributes
tt_pattern_callback_add(p, do_edit_message);
tt_pattern_register(p);
```

Note – Callbacks are called in reverse order of registration (for example, the most recently added callback is called first).

Figure 11-1 illustrates how the ToolTalk service invokes message and pattern callbacks when `tt_message_receive` is called to retrieve a new message.

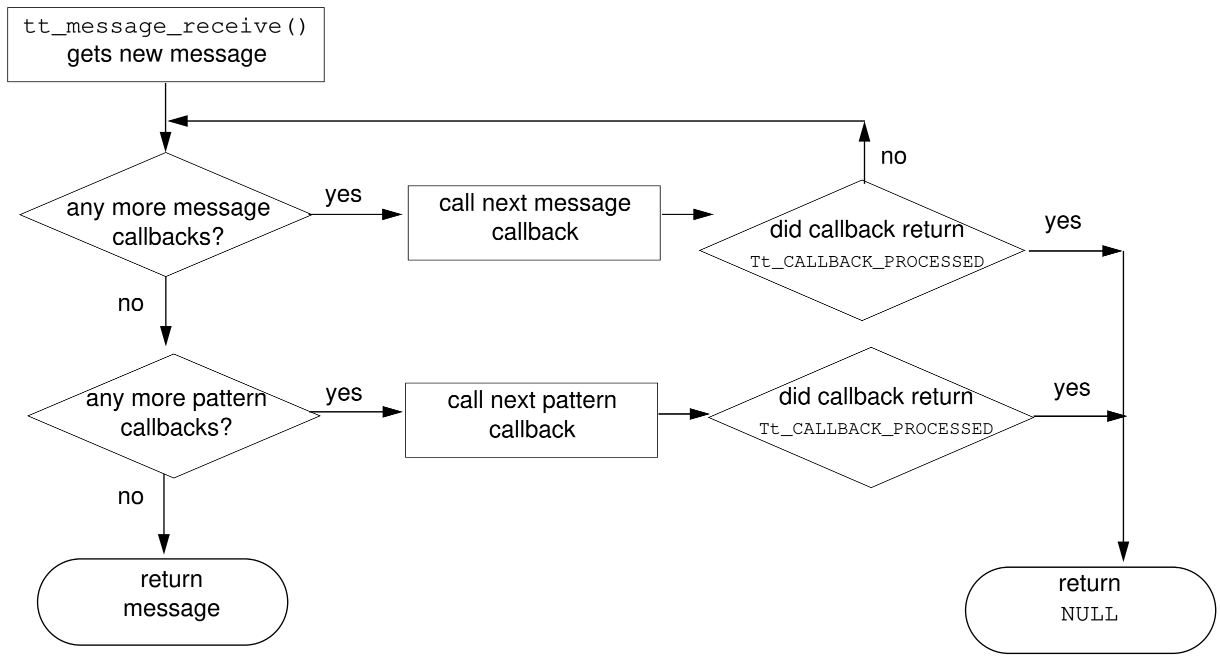

Figure 11-1 How Callbacks Are Invoked

Callbacks for Messages Addressed to Handlers

After the ToolTalk service determines the receiver for a message addressed to a handler, it matches the message against any patterns registered by the receiver. (Messages explicitly addressed to handlers are *point-to-point* messages and do not use pattern matching.)

- If the message does not match a pattern, the message is delivered in the normal manner.
- If the message is matched to a pattern, any callbacks attached to the pattern are run.

Attaching Callbacks to Static Patterns

Numeric tags (opnums) can be attached to each signature in a ptype when a static pattern is created. A callback can now be attached to the opnum. When a message is delivered because it matched a static pattern with an opnum, the ToolTalk service checks for any callbacks attached to the opnum and, if any exists, runs them.

Handling Requests

When your process receives a request (class = `TT_REQUEST`), you must either reply to the request, or reject or fail the request.

Replying to Requests

When you reply to a request, you need to:

1. Perform the requested operation.
2. Fill in any argument values with modes of `TT_OUT` or `TT_INOUT`.
3. Send the reply to the message.

Table 11-2 lists the ToolTalk functions you use to reply to requests.

Table 11-2 Functions to Reply to Requests

ToolTalk Function	Description
tt_message_arg_mode(Tt_message m, int n)	The argument mode (in, out, inout). †
tt_message_arg_bval_set(Tt_message m, int n, const unsigned char *value, int len)	Sets an argument's value to the specified byte array. ‡
tt_message_arg_ival_set(Tt_message m, int n, int value)	Sets an argument's value to the specified integer.‡
tt_message_arg_val_set(Tt_message m, int n, const char *value)	Sets an argument's value to the specified string. ‡
tt_message_arg_xval_set(Tt_message m, int n, xdrproc_t xdr_proc, void *value)	‡

† Return type is `Tt_mode`.
‡ Return type is `Tt_status`.

Table 11-2 Functions to Reply to Requests *(Continued)*

ToolTalk Function	Description
tt_message_context_set(Tt_message m, const char *slotname, const char *value);	Sets a context to the specified string. ‡
tt_message_bcontext_set(Tt_message m, const char *slotname, unsigned char *value, int length);	Sets a context to the specified byte array. ‡
tt_message_icontext_set(Tt_message m, const char *slotname, int value);	Sets a context to the specified integer.‡
tt_message_xcontext_set(Tt_message m, const char *slotname, xdrproc_t xdr_proc, void *value)	Sets a context to the specific xdr.‡
tt_message_reply(Tt_message m)	Replies to message. ‡

† Return type is `Tt_mode`.
‡ Return type is `Tt_status`.

Rejecting or Failing a Request

If you have examined the request and your application is not currently able to handle the request, you can use the ToolTalk functions listed in Table 11-3 to reject or fail a request.

Table 11-3 Rejecting or Failing Requests

ToolTalk Function	Description
tt_message_reject(Tt_message m)	Rejects message.
tt_message_fail(Tt_message m)	Fails message.
tt_message_status_set(Tt_message m, int status)	Sets the status of the message; this status is seen by the receiving application.
tt_message_status_string_set(Tt_message m, const char *status_str)	Sets the text that describes the status of the message; this text is seen be the receiving application.

Return type for these requests is `Tt_status`.

Rejecting a Request

If you have examined the request and your application is not currently able to perform the operation but another application might be able to do so, use `tt_message_reject` to reject the request.

When you reject a request, the ToolTalk service attempts to find another receiver to handle it. If the ToolTalk service cannot find a handler that is currently running, it examines the disposition attribute, and either queues the message or attempts to start applications with ptypes that contain the appropriate message pattern.

Failing a Request

If you have examined the request and the requested operation cannot be performed by you or any other process with the same ptype as yours, use `tt_message_fail` to inform the ToolTalk service that the operation cannot be performed. The ToolTalk service will inform the sender that the request failed.

To inform the sender of the reason the request failed, use `tt_message_status_set` or `tt_message_status_string_set` before you call `tt_message_fail`.

Note – The status code you specify with `tt_message_status_set` must be greater than `TT_ERR_LAST`.

Destroying Messages

After you have processed a message and no longer need the information in the message, use `tt_message_destroy` to delete the message and free storage space.

Objects 12

This chapter describes how to create ToolTalk specs for objects your application creates and manages. Before you can identify the type of objects, you need to define otypes and store them in the ToolTalk Types Database. See Chapter 10, “Static Message Patterns“ for more information on otypes.

The ToolTalk service uses spec and otype information to determine object-oriented message recipients.

See the demo programs in the following directory for an example of ToolTalk’s object-oriented messaging: `/usr/dt/examples/tt/edit_demo`

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

Object-Oriented Messaging

Object-oriented messages are addressed to objects managed by applications. To use object-oriented messaging, you need to be familiar with process-oriented messaging concepts and the ToolTalk concept of object.

Object Data

Object data is stored in two parts as shown in Figure 12-1.

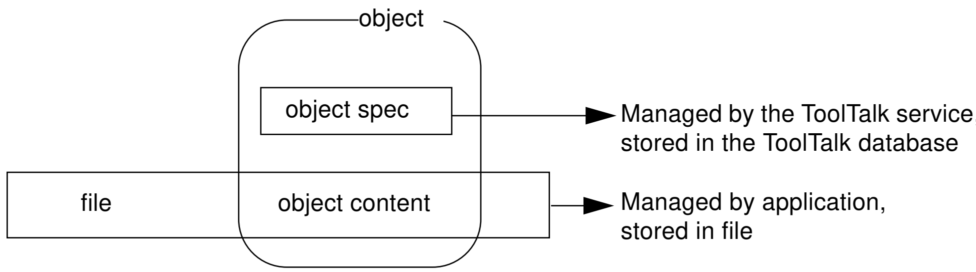

Figure 12-1 ToolTalk Object Data

One part is called the *object content*. The object content is managed by the application that creates or manages the object and is typically a piece, or pieces, of an ordinary file, for example, a paragraph, a source code function, or a range of spreadsheet cells.

The second part is called the *object specification* (*spec*). A spec contains standard properties such as the type of object, the name of the file in which the object contents are located, and the object owner. Applications can also add their own properties to a spec, for example, the location of the object content within a file. Because applications can store additional information in specs, you can identify data in existing files as objects without changing the formats of the files. You can also create objects from pieces of read-only files. Applications create and write specs to the ToolTalk database managed by `rpc.ttdbserverd`.

Note – You cannot create objects in files that reside in a read-only file system. The ToolTalk service must be able to create a database in the same file system that contains the object.

A *ToolTalk object* is a portion of application data for which a ToolTalk spec has been created.

Creating Object Specs

To instruct the ToolTalk service to deliver messages to your objects, you create a spec that identifies the object and its otype. Table 12-1 lists the ToolTalk functions you use to create and write object spec.

Table 12-1 Functions to Create

ToolTalk Function	Description
`tt_spec_create(const char *filepath)`	Creates spec. †
`tt_spec_prop_set(const char *objid, const char *propname, const char *value)`	Sets property to specified string value.‡
`tt_spec_prop_add(const char *objid, const char *propname, const char *value)`	Adds string property. ‡
`tt_spec_bprop_add(const char *objid, const char *propname, const unsigned char *value, int length)`	Adds byte array property. ‡
`tt_spec_bprop_set(const char *objid, const char *propname, const unsigned char *value, int length)`	Sets property to specified byte array value. ‡
`tt_spec_type_set(const char *objid, const char *otid)`	Sets object type of spec. ‡
`tt_spec_write(const char *objid)`	Writes spec to database.‡

† Return type is `char*`.
‡ Return type is `Tt_status`.

To create an object spec in memory and obtain an objid for the object, use `tt_spec_create`.

Assigning Otypes

To assign an otype for the object spec, use `tt_spec_type_set`. You must set the type before the spec is written for the first time. It cannot be changed.

Note – If you create an object spec without assigning an otype or with an otype that is unknown to the ToolTalk Types Database, messages addressed to the object cannot be delivered. (The ToolTalk service does not verify that the otype you specified is known to the ToolTalk Types Database.)

Determining Object Specification Properties

You can determine what *properties* you want associated with an object by adding these properties to a spec. The ToolTalk service recognizes that it is not always possible to store information in your own internal data, for example, the objid for objects in plain ASCII text files. You can store the location of the objid in a spec property and then use this location to identify where the object is in your tool's internal data structures.

The spec properties are also a convenience for the user. Users may want to associate properties (such as a comment or object name) with the object that they can view later. Your application or another ToolTalk-based tool can search for and display these properties for users.

Storing Spec Properties

To store properties in a `spec`, use `tt_spec_prop_set`.

Adding Values to Properties

To add to the list of values associated with the property, use `tt_spec_prop_add`.

Writing Object Specs

After you set the otype and add properties to an object spec, use `tt_spec_write` to make it a permanent ToolTalk item and visible to other applications. When you call `tt_spec_write`, the ToolTalk service writes the spec into the ToolTalk database.

Updating Object Specs

To update existing object spec properties, use `tt_spec_prop_set` and `tt_spec_prop_add` specifying the objid of the existing spec. Once the spec properties are updated, use `tt_spec_write` to write the changes into the ToolTalk database.

When you are updating an existing spec and the ToolTalk service returns `TT_WRN_STALE_OBJID` when you call `tt_spec_write`, it has found a forwarding pointer to the object in the ToolTalk database that indicates the object has been moved. To obtain the new objid, create an object message that contains the old objid and send it. The ToolTalk service will return the same status code, `TT_WRN_STALE_OBJID`, but updates the message objid attribute to contain the new objid. Use `tt_message_object` to retrieve the new objid from the message and put the new objid into your internal data structure.

Maintaining Object Specs

The ToolTalk service provides the functions to examine, compare, query, and move object specs. Table 12-2 lists the ToolTalk functions you use to maintain object specs.

Table 12-2 Functions to Maintain Object Specifications

Return Type	ToolTalk Function	Description
`char *`	`tt_spec_file(const char *objid)`	The name of the file on which the spec is located.
`char *`	`tt_spec_type(const char *objid)`	The object type of the spec.
`char *`	`tt_spec_prop(const char *objid, const char *propname, int i)`	Retrieves the *i*th (zero-based) property value as a string.
`int`	`tt_spec_prop_count(const char *objid, const char *propname)`	The number of values under this property name.

Table 12-2 Functions to Maintain Object Specifications *(Continued)*

Return Type	ToolTalk Function	Description
`Tt_status`	`tt_spec_bprop(const char *objid, const char *propname, int i, unsigned char **value, int *length)`	The number of byte array values under this property name.
`char *`	`tt_spec_propname(const char *objid, int i)`	The name of the *ith* property.
`int`	`tt_spec_propnames_count(const char *objid)`	The number of properties located on this spec.
`char *`	`tt_objid_objkey(const char *objid)`	The unique key of the spec id.
`Tt_status`	`tt_file_objects_query(const char *filepath, Tt_filter_function filter, void *context, void *accumulator)`	Queries the database for object specs.
`int`	`tt_objid_equal(const char *objid1, const char *objid2)`	Checks whether two spec ids are the same.
`char *`	`tt_spec_move(const char *objid, const char *newfilepath)`	Moves object spec to a new file.

Examining Spec Information

You can examine the following spec information with the specified ToolTalk functions:

- Path name of the file that contains the object: `tt_spec_file`
- Otype of this object: `tt_spec_type`
- Properties stored on the spec: `tt_spec_prop` or `tt_spec_bprop`

Comparing Object Specs

To compare two objids, use `tt_objid_equal`. `tt_objid_equal` returns a value of 1 even in the case where one objid is a forwarding pointer for the other.

Querying for Specific Specs in a File

Create a filter function to query for specific specs in a file and obtain the specs in which you are interested.

Use `tt_file_objects_query` to find all the objects in the named file. As the ToolTalk service finds each object, it calls your filter function, and passes it the objid of the object and the two application-supplied pointers. Your filter function does some computation and returns a `Tt_filter_action` value (`TT_FILTER_CONTINUE` or `TT_FILTER_STOP`) to either continue the query, or to quit the search and return immediately.

The following illustrates how to obtain a list of specs:

```
/*
 * Called to update the scrolling list of objects for a file. Uses
 * tt_file_objects_query to find all the ToolTalk objects.
 */
int
cntl_update_obj_panel()
{
       static int list_item = 0;
    char *file;
    int i;

    cntl_objid = (char *)0;

    for (i = list_item; i >= 0; i--) {
        /*Insert code to delete an item from the list*/
    }

    list_item = 0;
    file = XmTextGetString(cntl_ui_file_field);
    if (tt_file_objects_query(file,
                           (Tt_filter_function)cntl_gather_specs,
                                 &list_item, NULL) != TT_OK) {
        /*Insert code to post an error message*/
        return 0;
    }

    return 1;
}
```

Moving Object Specs

The objid contains a pointer to a particular file system where the spec information is stored. To keep spec information as available as the object described by the spec, the ToolTalk service stores the spec information on the same file system as the object. Therefore, if the object moves, the spec must move, too.

Use `tt_spec_move` to notify the ToolTalk service when an object moves from one file to another (for example, through a cut and paste operation).

- If a new objid is not required (because both the new and old files are in the same file system), the ToolTalk service returns TT_WRN_SAME_OBJID.
- If the object moved to another file system, the ToolTalk service returns a new objid for the object and leaves a forwarding pointer in the ToolTalk database from the old objid to the new one.

When your process sends a message to an out-of-date objid (that is, one with a forwarding pointer), tt_message_send returns a special status code, TT_WRN_STALE_OBJID, and replaces the object attribute in the message with a new objid that points to the same object in the new location.

Note – Update any internal data structures that reference the object with the new objid.

Destroying Object Specs

Use tt_spec_destroy to immediately destroy an object's spec.

Managing Object and File Information

Caution – Despite the efforts of the ToolTalk service and integrated applications, object references can still be broken if you remove, move, or rename files with standard operating system commands such as rm or mv. Broken references will result in undeliverable messages.

Managing Files that Contain Object Data

To keep up-to-date the ToolTalk database that services the disk partition where a file that contains object data is stored, use the ToolTalk functions to copy, move, or destroy the file. Table 12-3 lists the ToolTalk functions you use to manage files that contain object data.

Table 12-3 Functions to Copy, Move, or Remove Files that Contain Object Data

ToolTalk Function	Description
`tt_file_move(const char *oldfilepath, const char *newfilepath)`	Moves the file and the ToolTalk object data.
`tt_file_copy(const char *oldfilepath, const char *newfilepath)`	Copies the file and the ToolTalk object data.
`tt_file_destroy(const char *filepath)`	Removes the file and the ToolTalk object data.

The return type for these functions is `Tt_status`.

Managing Files that Contain ToolTalk Information

The ToolTalk service provides ToolTalk-enhanced shell commands to copy, move, and remove ToolTalk object and file information. Table 12-4 lists the ToolTalk-enhanced shell commands that you and users of your application should use to copy, move, and remove files referenced in messages and files that contain objects.

Table 12-4 ToolTalk-Wrapped Shell Commands

Command	Description
`ttcp`	Copies file to new location. Updates file and object location information in ToolTalk database.
`ttmv`	Renames directory or files. Updates file and object location information in ToolTalk database.

Table 12-4 ToolTalk-Wrapped Shell Commands

Command	Description
`ttrm`	Removes specified file. Removes file and object information from the ToolTalk database.
`ttrmdir`	Removes empty directories (directories that contain no files) that have ToolTalk object specs associated with them. (It is possible to create an object spec for a directory; when an object spec is created, the path name of a file or directory is supplied.) Removes object information from the ToolTalk database.
`tttar`	Archives (or extracts) multiple files and object information into (or from) a single archive, called a tarfile. Can also be used to only archive (or extract) ToolTalk file and object information into (or from) a tarfile.

An Example of Object-Oriented Messaging

You can run the edit_demo program for a demonstration of ToolTalk object-oriented messaging. This demo consists of two programs – `cntl` and `edit`. The cntl program uses the ToolTalk service to start an edit process with which to edit a specified file; the edit program allows you to create ToolTalk objects and associate the objects with text in the file. Once objects have been created and associated with text, you can use the cntl program to query the file for the objects and to send messages to the objects.

The sample code creates an object for its user: it creates the object spec, sets the otype, writes the spec to the ToolTalk database, and wraps the user's selection with C-style comments. The application also sends out a procedure-addressed notice after it creates the new object to update other applications that observe messages with the `CDE_EditDemo_new_object` operation. If other applications are displaying a list of objects in a file managed by `CDE_EditDemo`, they update their list after receiving this notice.

Managing Information Storage 13

To simplify your application storage management, the ToolTalk service copies all information your application provides to the ToolTalk service and also provides you with a copy of the information it returns to your application.

Information Provided to the ToolTalk Service

When you provide a pointer to the ToolTalk service, the information referenced by the pointer is copied. You can then dispose of the information you provided; the ToolTalk service will not use the pointer again to retrieve the information.

Information Provided by the ToolTalk Service

The ToolTalk service provides an allocation stack in the ToolTalk API library to store information it gives to you. For example, if you ask for the sessid of the default session with `tt_default_session`, the ToolTalk service returns the address of the character string in the allocation stack (a `char *` pointer) that contains the sessid. After you retrieve the sessid, you can dispose of the character string to clean up the allocation stack.

Note – Do not confuse the API allocation stack with your program's runtime stack. The API stack will not discard information until instructed to do so.

Calls Provided to Manage the Storage of Information

The ToolTalk service provides the calls listed in Table 13-1 to manage the storage of information in the ToolTalk API allocation stack:

Table 13-1 Managing ToolTalk Storage

Return Type	ToolTalk Function	Description
`int`	`tt_mark(void)`	Marks information returned by a series of functions.
`void`	`tt_release(int mark)`	Frees information returned by a series of functions.
`caddr_t`	`tt_malloc(size_t s)`	Reserves a specified amount of storage in the allocation stack for your use.
`void`	`tt_free(caddr_t p)`	Frees storage set aside by `tt_malloc`. This function takes an address returned by the ToolTalk API and frees the associated storage.

Marking and Releasing Information

The `tt_mark` and `tt_release` functions are a general mechanism to help you easily manage information storage. The `tt_mark` and `tt_release` functions are typically used at the beginning and end of a routine where the information returned by the ToolTalk service is no longer necessary once the routine has ended.

Marking Information for Storage

To ask the ToolTalk service to mark the beginning of your storage space, use `tt_mark`. The ToolTalk service returns a mark, an integer that represents a location on the API stack. All the information that the ToolTalk service subsequently returns to you will be stored in locations that come after the mark.

Releasing Information No Longer Needed

When you no longer need the information contained in your storage space, use `tt_release` and specify the mark that signifies the beginning of the information you no longer need.

Example of Marking and Releasing Information

The following code sample calls `tt_mark` at the beginning of a routine that examines the information in a message. When the information examined in the routine is no longer needed and the message has been destroyed, `tt_release` is called with the mark to free storage on the stack.

```
/*
 * Get a storage mark so we can easily free all the data
 * ToolTalk returns to us.
 */

mark = tt_mark();

if (0==strcmp("ttsample1_value", tt_message_op(msg_in))) {
        tt_message_arg_ival(msg_in, 0, &val_in);
}

tt_message_destroy(msg_in);
tt_release(mark);
return;
```

Allocating and Freeing Storage Space

The `tt_malloc` and `tt_free` functions are a general mechanism to help you easily manage storage allocation.

Allocating Storage Space

`tt_malloc` reserves a specified amount of storage in the allocation stack for your use. For example, you can use `tt_malloc` to create a storage location and copy the sessid of the default session into that location.

Freeing Allocated Storage Space

To free storage of individual objects that the ToolTalk service provides pointers to, use `tt_free`. For example, you can free up the space in the API allocation stack that stores the sessid after you have examined the sessid. `tt_free` takes an address in the allocation stack (a `char *` pointer or an address returned from `tt_malloc`) as an argument.

Special Case: Callback and Filter Routines

The way that the ToolTalk service behaves toward information passed into filter functions and callbacks is a special case. Callback and filter routines called by the ToolTalk service are called with two kinds of arguments:

- Context arguments — the arguments you passed into the API call that triggered the callback. These arguments point to items owned by your application.
- Pointers to API objects — the address of message or pattern attributes in storage.

The context arguments are passed from the ToolTalk service to your application. The API objects referenced by pointers are freed by the ToolTalk service as soon as your callback or filter function returns. If you want to keep any of these objects, you must copy the objects before your function returns.

Note – The way that the ToolTalk service behaves toward information passed into filter functions and callbacks is a special case. In all other instances, the ToolTalk service stores the information in the API allocation stack until you free it.

Callback Routines

One of the features of the ToolTalk service is callback support for messages, patterns, and filters. Callbacks are routines in your program that ToolTalk calls when a particular message arrives (*message callback*) or when a message matches a particular pattern you registered (*pattern callback*).

To tell the ToolTalk service about these callbacks, add the callback to a message or pattern before you send the message or register the pattern.

Filter Routines

When you call file query functions such as tt_file_objects_query, you point to a filter routine that the ToolTalk service calls as it returns items from the query. For example, you could use filter routine used by the ToolTalk file query function to find a specific object. The tt_file_objects_query function returns all the objects in a file and runs the objects through a filter routine that you provide. Once your filter routine finds the specified object, you can use tt_malloc to create a storage location and copy the object into the location. When your filter function returns, the ToolTalk service will free all storage used by the objects in the file but the object you stored with the tt_malloc call will be available for further use.

Handling Errors 14

The ToolTalk service returns error status in the function's return value rather than in a global variable. ToolTalk functions return one of these error values:

- `Tt_status`
- `int`
- `char*` or opaque handle

Each return type is handled differently to determine if an error occurred. For example, the return value for `tt_default_session_set` is a `Tt_status` code. If the ToolTalk service sets the default session to the specified sessid:

- Without a problem — the `Tt_status` code returned is `TT_OK`.
- With a problem — the `Tt_status` code returned is `TT_ERR_SESSION`. This status code informs you that the sessid you passed was not valid.

Retrieving ToolTalk Error Status

You can use the ToolTalk error handling functions shown in Table 14-1 to retrieve error values.

Table 14-1 Retrieving ToolTalk Error Status

ToolTalk Function	Description
tt_pointer_error(char * return_val)	Returns an error encoded in a pointer.
tt_pointer_error((void *) (p))	Returns an error encoded in a pointer cast to `VOID *`.
tt_int_error(int return_val)	Returns an error encoded in an integer.
The return type for these functions is `Tt_status`.	

Checking ToolTalk Error Status

You can use the ToolTalk error macro shown in Table 14-2 to check error values.

Table 14-2 ToolTalk Error Macros

Return Type	ToolTalk Macro	Expands to
`Tt_status`	`tt_is_err(status_code)`	(`TT_WRN_LAST` < (status_code))

Returned Value Status

Functions with Natural Return Values

If a ToolTalk function has a natural return value such as a pointer or an integer, a special *error value* is returned instead of the real value.

Functions with No Natural Return Values

If a ToolTalk function does not have a natural return value, the return value is an element of `Tt_status` *enum*.

To see if there is an error, use the ToolTalk macro `tt_is_err`, which returns an integer.

- If the return value is 0, the `Tt_status` *enum* is either `TT_OK` or a warning.
- If the return value is 1, the `Tt_status` *enum* is an error.

If there is an error, you can use the `tt_status_message` function to obtain the character string that explains the `Tt_status` code, as shown below.

```
char *spec_id, my_application_name;
Tt_status tterr;

tterr = tt_spec_write(spec_id);
if (tt_is_err(tterr)) {
    fprintf(stderr, "%s: %s\n", my_application_name,
        tt_status_message(tterr));
}
```

Returned Pointer Status

If an error occurs during a ToolTalk function that returns a pointer, the ToolTalk service provides an address within the ToolTalk API library that indicates the appropriate `Tt_status` code. To check whether the pointer is valid, you can use the ToolTalk macro `tt_ptr_error`. If the pointer is an error value, you can use `tt_status_message` to get the `Tt_status` character string.

The following checks the pointer and retrieves and prints the `Tt_status` character string if an error value is found:

```
char *old_spec_id, new_file, new_spec_id, my_application_name;
Tt_status tterr;

new_spec_id = tt_spec_move(old_spec_id, new_file);
tterr = tt_ptr_error(new_spec_id);
switch (tterr) {
    case TT_OK:
    /*
     * Replace old_spec_id with new_spec_id in my internal
     * data structures.
     */
    break;
    case TT_WRN_SAME_OBJID:
    /*
```

```
         * The spec must have stayed in the same filesystem,
         * since ToolTalk is reusing the spec id. Do nothing.
         */
        break;
        case TT_ERR_FILE:
        case TT_ERR_ACCESS:
        default:
        fprintf(stderr, "%s: %s\n", my_application_name,
            tt_status_message(tterr));
        break;
}
```

Returned Integer Status

If an error occurs during a ToolTalk function that returns an integer, the return value is out-of-bounds. The `tt_int_error` function returns a status of `TT_OK` if the value is not out-of-bounds.

To check if a value is out-of-bounds, you can use the `tt_is_err` macro to determine if an error or a warning occurred.

To retrieve the character string for a `Tt_status` code, you can use `tt_status_message`.

The example below checks a returned integer:

```
Tt_message msg;
int num_args;
Tt_status tterr;
char *my_application_name;

num_args = tt_message_args_count(msg);
tterr = tt_int_error(num_args);
if (tt_is_err(tterr)) {
    fprintf(stderr, "%s: %s\n", my_application_name,
        tt_status_message(tterr));
}
```

Broken Connections

The ToolTalk service provides a function to notify processes if your tool exits unexpectedly. When you include the `tt_message_send_on_exit` call, the ToolTalk service queues the message internally until one of two events happen:

1. Your process calls `tt_close`.

   In this case, the ToolTalk service deletes the message from its queue.

2. The connection between the ttsession server and your process is broken; for example, the application crashed.

   In this case, the ToolTalk service matches the queued message to a pattern and delivers it in the same manner as if you had sent the message normally before exiting.

Your process can also send a normal message on a normal termination by calling `tt_message_send` before it calls `tt_close`. In this case, if your process sends its normal termination message but crashes before it calls `tt_close`, the ToolTalk service will deliver both the normal termination message and the `tt_message_send_on_exit` message to interested processes.

Error Propagation

ToolTalk functions that accept pointers always check the pointer passed in and return `TT_ERR_POINTER` if the pointer is an error value. This check allows you to combine calls in reasonable ways without checking the value of the pointer for every single call.

In the following code sample, a message is created, filled in, and sent. If `tt_message_create` fails, an error object is assigned to *m*, and all the `tt_message_`*xxx*`_set` and `tt_message_send` calls fail. To detect the error without checking between each call, you only need to check the return code from `tt_message_send`.

```
Tt_message m;

m=tt_message_create();
tt_message_op_set(m,"OP");
tt_message_address_set(m,TT_PROCEDURE);
tt_message_scope_set(m,TT_SESSION);
tt_message_class_set(m,TT_NOTICE);
tt_rc=tt_message_send(m);
if (tt_rc!=TT_OK)...
```

Initialization Error Messages

The ToolTalk error messages described in Table 14-3 can occur either when the ToolTalk service, or an application that uses the ToolTalk service, is attempting to start up.

Table 14-3 Errors that may Occur During Initialization

Error Message	Description	Solution
`ld: libtt*: not found`	The runtime linker could not find the dynamic library.	Place a directory that contains the dynamic library in the the shared library search path.

Table 14-3 Errors that may Occur During Initialization *(Continued)*

Error Message	Description	Solution
`/usr/bin/sh: application_name: not found`	The start string as installed in the ToolTalk Types Database does not correspond to an executable file in `$PATH`.	To correct this error: a. First, start the application as it would be started without the ToolTalk service. b. After the application has started, retry the operation that should have started the application with the ToolTalk service. To prevent the error from occurring again, verify that the start string in the relevant ptype corresponds to an executable file in `$PATH`.
`Cannot open display`	`ttsession` could not contact the server named with the `-d` *`display`* option or the `$DISPLAY` variable.	To start the session: a. Verify that the named display is running. b. Verify that the host on which `ttsession` is running has permission to connect to the named display.
`ttsession: Illegal environment (-c or -d not specified and DISPLAY variable not set)`	Neither the `-d` *`display`* option or the `$DISPLAY` variable is set. This error typically occurs when you or your client attempt to start `ttsession` after you have either switched to another user name or become superuser.	Set the `-d` *`display`* option or the `$DISPLAY` variable.

ToolTalk Error Messages

The ToolTalk error and warning identifiers are allocated as follows:

TT_OK	TT_WRN_*	APP_WRN_*	TT_WRN_LAST	TT_ERR_*	APP_ERR_*	TT_ERR_LAST
0	1	512	1024	1025	1536	2047

Table 14-4 is an alphabetical listing of the ToolTalk error messages and their corresponding message ids.

Table 14-4 Alphabetical List of ToolTalk Error Messages

Error Message	Message ID
TT_ERR_ACCESS	TTERR-1032
TT_ERR_ADDRESS	TTERR-1039
TT_ERR_APPFIRST	TTERR-1536
TT_ERR_CATEGORY	TTERR-1057
TT_ERR_CLASS	TTERR-1025
TT_ERR_DBAVAIL	TTERR-1026
TT_ERR_DBCONSIST	TTERR-1060
TT_ERR_DBEXIST	TTERR-1027
TT_ERR_DBFULL	TTERR-1059
TT_ERR_DBUPDATE	TTERR-1058
TT_ERR_DISPOSITION	TTERR-1046
TT_ERR_FILE	TTERR-1028
TT_ERR_INTERNAL	TTERR-1051
TT_ERR_LAST	TTERR-2047
TT_ERR_MODE	TTERR-1031

Table 14-4 Alphabetical List of ToolTalk Error Messages

Error Message	Message ID
TT_ERR_NO_MATCH	TTERR-1053
TT_ERR_NOMEM	TTERR-1062
TT_ERR_NOMP	TTERR-1033
TT_ERR_NOTHANDLER	TTERR-1034
TT_ERR_NO_VALUE	TTERR-1050
TT_ERR_NUM	TTERR-1035
TT_ERR_OBJID	TTERR-1036
TT_ERR_OP	TTERR-1037
TT_ERR_OTYPE	TTERR-1038
TT_ERR_OVERFLOW	TTERR-1055
TT_ERR_PATH	TTERR-1040
TT_ERR_POINTER	TTERR-1041
TT_ERR_PROCID	TTERR-1042
TT_ERR_PROPLEN	TTERR-1043
TT_ERR_PROPNAME	TTERR-1044
TT_ERR_PTYPE	TTERR-1045
TT_ERR_PTYPE_START	TTERR-1056
TT_ERR_READONLY	TTERR-1052
TT_ERR_SCOPE	TTERR-1047
TT_ERR_SESSION	TTERR-1048
TT_ERR_SLOTNAME	TTERR-1063
TT_ERR_STATE	TTERR-1061

Table 14-4 Alphabetical List of ToolTalk Error Messages

Error Message	Message ID
TT_ERR_UNIMP	TTERR-1054
TT_ERR_VTYPE	TTERR-1049
TT_ERR_XDR	TTERR-1064
TT_OK	TTERR-0
TT_STATUS_LAST	TTERR-2048
TT_WRN_APPFIRST	TTERR-512
TT_WRN_LAST	TTERR-1024
TT_WRN_NOTFOUND	TTERR-1
TT_WRN_SAME_OBJID	TTERR-4
TT_WRN_STALE_OBJID	TTERR-2
TT_WRN_START_MESSAGE	TTERR-5
TT_WRN_STOPPED	TTERR-3

Table 14-5 describes the ToolTalk error messages; the error messages are listed in order of their message id.

Table 14-5 ToolTalk Error Messages

Error Message	Message ID	Error Message String	Description	Solution
TT_OK	TTERR-0	TT_OK Request successful.	The call was completed successfully.	
TT_WRN_NOTFOUND	TTERR-1	TT_WRN_NOTFOUND The object was not removed because it was not found.	The ToolTalk service could not find the specified object in the ToolTalk database. The destroy operation did not succeed.	

Table 14-5 ToolTalk Error Messages *(Continued)*

Error Message	Message ID	Error Message String	Description	Solution
`TT_WRN_STALE_OBJID`	TTERR-2	`TT_WRN_STALE_OBJID` The object attribute in the message has been replaced with a newer one. Update the place from which the object id was obtained.	When the ToolTalk service looked up the specified object in the ToolTalk database, it found a forwarding pointer to the object.	The ToolTalk service automatically puts the new objid in the message. a. Use `tt_message_object()` to retrieve the new objid. b. Update any internal application references to the new objid.
`TT_WRN_STOPPED`	TTERR-3	`TT_WRN_STOPPED` The query was halted by the filter procedure.	The query operation being performed was halted by the Tt_filter_function.	
`TT_WRN_SAME_OBJID`	TTERR-4	`TT_WRN_SAME_OBJID` The moved object retains the same objid.	The object moved stayed within the same file system. The ToolTalk service will retain the same objid and update the location.	
`TT_WRN_START_MESSAGE`	TTERR-5	`TT_WRN_START_MESSAGE` This message caused this process to be started. This message should be replied to even if it is a notice.	When the ToolTalk service starts an application to deliver a message to it, a reply to that message must be sent even if the message which ToolTalk is attempting to deliver is a notice.	Use `tt_message_accept()` or `tt_message_reply()` to reply to, fail, or reject the message after the process is started by the ToolTalk service.
`TT_WRN_APPFIRST`	TTERR-512	`TT_WRN_APPFIRST` This code should be unused.	This code marks the beginning of the messages allocated for ToolTalk application warnings.	
`TT_WRN_LAST`	TTERR-1024	`TT_WRN_LAST` This code should be unused.	This code marks the last of the messages allocated for ToolTalk warnings.	
`TT_ERR_CLASS`	TTERR-1025	`TT_ERR_CLASS` The Tt_class value passed is invalid.	The ToolTalk service does not recognize the class value specified.	The Tt_class values are `TT_NOTICE` and `TT_REQUEST`. Retry the call with one of these values.
`TT_ERR_DBAVAIL`	TTERR-1026	`TT_ERR_DBAVAIL` A required database is not available. The condition may be temporary, trying again later may work.	The ToolTalk service could not access the ToolTalk database needed for this operation.	a. Check if the file server or workstation that contains the database is available. b. Try the operation again later.

Table 14-5 ToolTalk Error Messages *(Continued)*

Error Message	Message ID	Error Message String	Description	Solution
`TT_ERR_DBEXIST`	TTERR-1027	`TT_ERR_DBEXIST` A required database does not exist. The database must be created before this action will work.	The ToolTalk service did not find the specified ToolTalk database in the expected place.	Install the rpc.ttdbserve program on the machine that stores the file or object involved in this operation.
`TT_ERR_FILE`	TTERR-1028	`TT_ERR_FILE` File object could not be found.	The file specified does not exist or is not accessible.	a. Check the file path name and retry the operation. b. Check if the machine where the file is stored is accessible.
`TT_ERR_MODE`	TTERR-1031	`TT_ERR_MODE` The Tt_mode value is not valid.	The ToolTalk service does not recognize the specified mode value.	The Tt_mode values are `TT_IN`, `TT_OUT`, and `TT_INOUT`. Retry the call with one of these values.
`TT_ERR_ACCESS`	TTERR-1032	`TT_ERR_ACCESS` An attempt was made to access a ToolTalk object in a way forbidden by the protection system.	You do not have the necessary access to the object and the application; for example, you do not have permission to destroy an object spec. Therefore, the operation cannot be performed.	a. Obtain proper access to the object. b. Retry the operation.
`TT_ERR_NOMP`	TTERR-1033	`TT_ERR_NOMP` No ttsession process is running, probably because tt_open() has not been called yet. If this code is returned from tt_open() it means ttsession could not be started, which generally means ToolTalk is not installed on this system.	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.	a. Verify that the ToolTalk service is installed. b. Verify that ttsession is installed on the machine in use.
`TT_ERR_NOTHANDLER`	TTERR-1034	`TT_ERR_NOTHANDLE` Only the handler of the message can do this.	Only the handler of a message can perform this operation. This application is not the handler for this message.	

Table 14-5 ToolTalk Error Messages *(Continued)*

Error Message	Message ID	Error Message String	Description	Solution
`TT_ERR_NUM`	TTERR-1035	`TT_ERR_NUM` The integer value passed is not valid.	An invalid integer value that was out-of-range was passed to the ToolTalk service. Note - Simple out-of-range conditions, such as requesting the third value of a property that has only two values, return a null value.	Check the integer specified.
`TT_ERR_OBJID`	TTERR-1036	`TT_ERR_OBJID` The object id passed does not refer to any existing object spec.	The objid does not reference an existing object.	Update the spec property that contains the objid specified.
`TT_ERR_OP`	TTERR-1037	`TT_ERR_OP` The operation name passed is not syntactically valid.	The specified operation name is null or contains non-alphanumeric characters.	a. Remove any non-alphanumeric characters. b. Retry the operation.
`TT_ERR_OTYPE`	TTERR-1038	`TT_ERR_OTYPE` The object type passed is not the name of an installed object type.	The ToolTalk service could not locate the specified otype.	Check the type of the object with `tt_spec_type()`. If the application was recently installed and the ToolTalk service has not reread the ToolTalk Types Database: a. Locate the process id for the ttsession. b. Force the reread with the USR-2 signal: % ps -elf | grep ttsession % kill -USR2 <*ttsession pid*>
`TT_ERR_ADDRESS`	TTERR-1039	`TT_ERR_ADDRESS` The Tt_address value passed is not valid.	The ToolTalk service does not recognize the address value specified.	The Tt_address values are `TT_PROCEDURE`, `TT_OBJECT`, `TT_HANDLER`, and `TT_OTYPE`. Retry the call with one of these values.
`TT_ERR_PATH`	TTERR-1040	`TT_ERR_PATH` One of the directories in the file path passed does not exist or cannot be read.	The ToolTalk service was not able to read a directory in the file path name.	a. Check the pathname to ensure access to the specified directories. b. Check the machine where the file resides to make sure it is accessible.

Table 14-5 ToolTalk Error Messages *(Continued)*

Error Message	Message ID	Error Message String	Description	Solution
TT_ERR_POINTER	TTERR-1041	TT_ERR_POINTER The opaque pointer (handle) passed does not indicate an object of the proper type.	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.	a. Check the arguments for the ToolTalk function to find what arguments the function expects. b. Retry the operation with a pointer for a valid object.
TT_ERR_PROCID	TTERR-1042	TT_ERR_PROCID The process id passed is not valid.	The process identifier specified is out of date or invalid.	Retrieve the default procid with `tt_default_procid()`.
TT_ERR_PROPLEN	TTERR-1043	TT_ERR_PROPLEN The property value passed is too long.	The ToolTalk service accepts property values of up to 64 characters.	Shorten the property value to less than 64 characters.
TT_ERR_PROPNAME	TTERR-1044	TT_ERR_PROPNAME The property name passed is syntactically invalid.	The property name is too long, contains non-alphanumeric characters, or is null.	Check the property name, modify if necessary, and retry the operation.

TT_ERR_PTYPE	TTERR-1045	TT_ERR_PTYPE The process type passed is not the name of an installed process type.	The ToolTalk service could not locate the specified ptype.	If the application was recently installed and the ToolTalk service has not reread the ToolTalk Types Database:
				a. Locate the process id for the ttsession.
				b. Force the reread with the USR-2 signal:
				`% ps -elf | grep ttsession`
				`% kill -USR2 <ttsession pid>`

TT_ERR_DISPOSITION	TTERR-1046	TT_ERR_DISPOSITION The Tt_disposition value passed is not valid.	The disposition passed is not recognized by the ToolTalk service.	The `Tt_disposition` values are `TT_DISCARD`, `TT_QUEUE`, and `TT_START`. Retry the call with one of these values.
TT_ERR_SCOPE	TTERR-1047	TT_ERR_SCOPE The Tt_scope value passed is not valid.	The scope passed is not recognized by the ToolTalk service.	The `Tt_scope` values are `TT_SESSION` and `TT_FILE`. Retry the call with one of these values.

Table 14-5 ToolTalk Error Messages *(Continued)*

Error Message	Message ID	Error Message String	Description	Solution
TT_ERR_SESSION	TTERR-1048	TT_ERR_SESSION The session id passed is not the name of an active session.	An out-of-date or invalid ToolTalk session was specified.	Either: a. obtain the sessid of the current default session using `tt_default_session()` b. obtain the sessid of the initial session in which the application was started using `tt_initial_session()`
TT_ERR_VTYPE	TTERR-1049	TT_ERR_VTYPE The value type name passed is not valid.	The specified property exists in the ToolTalk database but the type of value does not match the specified type; or the value type is not one that the ToolTalk service recognizes. The ToolTalk service supports types of int and string.	a. Change the type of the value to either int or string. b. Retry the operation.
TT_ERR_NO_VALUE	TTERR-1050	TT_ERR_NO_VALUE No property value with the given name and number exists.	The ToolTalk service could not locate a value for the property specified in the ToolTalk database.	Retrieve the current list of properties to find the property.
TT_ERR_INTERNAL	TTERR-1051	TT_ERR_INTERNAL Internal error (bug)	The ToolTalk service has suffered an internal error.	a. Restart all applications that are using the ToolTalk service. b. Report the error to the your system vendor support center.
TT_ERR_READONLY	TTERR-1052	TT_ERR_READONLY The attribute cannot be changed.	The application does not have ownership or write permissions for the attribute. Therefore, this operation cannot be performed.	
TT_ERR_NO_MATCH	TTERR-1053	TT_ERR_NO_MATCH No handler could be found for this message, and the disposition was not queue or start.	The message the application sent could not be delivered. No applications that are running have registered interest in this type of message.	Use `tt_disposition_set()` to change the disposition to TT_QUEUE or TT_START and resend the message. If no recipients are found, no application has registered interest in this type of message.

Table 14-5 ToolTalk Error Messages *(Continued)*

Error Message	Message ID	Error Message String	Description	Solution
TT_ERR_UNIMP	TTERR-1054	TT_ERR_UNIMP Function not implemented.	The ToolTalk function called is not implemented.	
TT_ERR_OVERFLOW	TTERR-1055	TT_ERR_OVERFLOW Too many active messages (try again later).	The ToolTalk service has received the maximum amount of active messages (2000) it can properly handle.	Either: a. Retrieve any messages that the ToolTalk service may be queueing for the application, and send the message again later. b. Start ttsession with the -A option. Specify the maximum number of messages in progress before a TT_ERR_OVERFLOW condition is returned. The default is 2000 messages.
TT_ERR_PTYPE_START	TTERR-1056	TT_ERR_PTYPE_START Attempt to launch a client specified in the start attribute of a ptype failed.	The ToolTalk service could not start the type of process specified.	Verify that the application that the ptype represents is properly installed and has execute permission.
TT_ERR_CATEGORY	TTERR-1057	TT_ERR_CATEGORY Pattern object has no category set.	The category was not set.	
TT_ERR_DBUPDATE	TTERR-1058	TT_ERR_DBUPDATE The database is inconsistent: another tt_spec_write updated object first.	The ToolTalk service could not update the database because the specified object was already updated by a previous tt_spec_write call.	
TT_ERR_DBFULL	TTERR-1059	ToolTalk database is full.	The ToolTalk service could not write to the database because it is full.	Create more space on the file system in which the database is stored.
TT_ERR_DBCONSIST	TTERR-1060	Database is access information is incomplete or database is corrupt (run ttdbck).	The ToolTalk service could not write to the database because it is either corrupt, or the access information is incomplete.	Run the ttdbck utility to repair the database.
TT_ERR_STATE	TTERR-1061	The Tt_message is in a state that is not valid for the attempted operation.	The state of the message is invalid for the type of operation being requested.	

Table 14-5 ToolTalk Error Messages *(Continued)*

Error Message	Message ID	Error Message String	Description	Solution
`TT_ERR_NOMEM`	TTERR-1062	No more memory.	There is not enough available memory to perform the operation.	Check the swap space, then retry the operation.
`TT_ERR_SLOTNAME`	TTERR-1063	The slot name is syntactically invalid.	The syntax for the slot name is not valid.	Correct the syntax for the slot name.
`TT_ERR_XDR`	TTERR-1065	The XDR procedure failed on the given data, or evaluated to a `0` length structure.	The XDR procedure failed on the given data, or evaluated to a `0` length structure.	
`TT_ERR_APPFIRST`	TTERR-1536	`TT_ERR_APPFIRST` This code should be unused.	This code marks the beginning of the messages allocated for ToolTalk application errors.	
`TT_ERR_LAST`	TTERR-2047	`TT_ERR_LAST` This code should be unused.	This code marks the last of the messages allocated for ToolTalk errors.	
`TT_STATUS_LAST`	TTERR-2048	`TT_STATUS_LAST` This code should be unused.	This code marks the last of the messages allocated for ToolTalk status.	

The ToolTalk Enumerated Types 15

This chapter provides reference information for the enumerated types component of the ToolTalk application programming interface (API).

The ToolTalk enumerated types fall into these categories:

- `Tt_address`
- `Tt_callback`
- `Tt_category`
- `Tt_class`
- `Tt_disposition`
- `Tt_filter`
- `Tt_mode`
- `Tt_scope`
- `Tt_state`
- `Tt_status`

They are listed in alphabetical order in each section.

Tt_address

`Tt_address` indicates which message attributes form the address to which the message will be delivered. Table 15-1 describes the possible values.

Table 15-1 Possible Values for `Tt_address`

Value	Description
`TT_HANDLER`	Addressed to a specific handler that can perform this operation with these arguments. Fill in *handler*, *op*, and arg attributes of the message or pattern.
`TT_OBJECT`	Addressed to a specific object that performs this operation with these arguments. Fill in *object*, *op*, and *arg* attributes of the message or pattern.
`TT_OTYPE`	Addressed to the type of object that can perform this operation with these arguments. Fill in *otype*, *op*, and *arg* attributes of the message or pattern.
`TT_PROCEDURE`	Addressed to any process that can perform this operation with these arguments. Fill in the *op* and *arg* attributes of the message or pattern.

`Tt_callback`

These values are used to specify the action taken by the callback attached to messages or patterns. If no callback returns `TT_CALLBACK_PROCESSED`, `tt_message_receive()` will return the message. Table 15-2 describes the possible values.

Table 15-2 Possible Values for `Tt_callback`

Value	Description
`TT_CALLBACK_CONTINUE`	If the callback returns `TT_CALLBACK_CONTINUE`, other callbacks will be run.
`TT_CALLBACK_PROCESSED`	If the callback returns `TT_CALLBACK_PROCESSED`, no further callbacks will be invoked for this event, and the message will not be returned by `tt_message_receive()`.

Tt_category

Tt_category values for the category attribute of a pattern indicate the receiver's intent. Table 15-3 describes the possible values.

Table 15-3 Possible Values for Tt_category

Value	Description
TT_OBSERVE	Just looking at the message. No feedback will be given to the sender.
TT_HANDLE	Will process the message, including filling in return values if any.

Tt_class

These values for the class attribute of a message or pattern indicate whether the sender wants an action to take place after the message has been received. Table 15-4 describes the possible values.

Table 15-4 Possible Values for Tt_class

Value	Description
TT_NOTICE	Notice of an event. Sender does not want feedback on this message.
TT_REQUEST	Request for some action to be taken. Sender must be notified of progress, success or failure, and must receive any return values.

Tt_disposition

Tt_disposition values indicate whether the receiving application should be started to receive the message or if the message should be queued until the receiving process is started at a later time. The message can also be discarded if the receiver is not started.

Note that Tt_disposition values can be added together, so that TT_QUEUE+TT_START means both to queue the message and to try to start a process. This can be useful if the start can fail (or be vetoed by the user) to ensure the message is processed as soon as an eligible process does start.

Table 15-5 describes the possible values.

Table 15-5 Possible Values for Tt_disposition

Value	Description
TT_DISCARD = 0	No receiver for this message. Message is returned to sender with the Tt_status field containing TT_FAILED.
TT_QUEUE = 1	Queue the message until a process of the proper ptype receives the message.
TT_START = 2	Attempt to start a process of the proper ptype if none is running.

Tt_filter

Tt_filter_action is the return value from a query callback filter procedure. Table 15-6 describes the possible values.

Table 15-6 Possible Values for Tt_filter

Value	Description
TT_FILTER_CONTINUE	Continue the query, feed more values to the callback.
TT_FILTER_STOP	Stop the query, don't look for any more values.

Tt_mode

Tt_mode values specify whether the sender, handler, or observers writes a message argument. Table 15-7 describes the possible values.

Table 15-7 Possible Values for Tt_mode

Value	Description
TT_IN	The argument is written by the sender and read by the handler and any observers.
TT_OUT	The argument is written by the handler and read by the sender and any reply observers.
TT_INOUT	The argument is written by the sender and the handler and read by all.

Tt_scope

Tt_scope values for the Scope attribute of a message or pattern indicate the set of processes eligible to receive the message. Table 15-8 describes the possible values.

Table 15-8 Possible Values for Tt_scope

Value	Description
TT_SESSION	All processes joined to the indicated session are eligible.
TT_FILE	All processes joined to the indicated file are eligible.
TT_BOTH	All processes joined to either the indicated file or the indicated session are eligible.
TT_FILE_IN_SESSION	All processes joined to both the indicated session and the indicated file are eligible.

Tt_state

Tt_state values indicate a message's delivery status. Table 15-9 describes the possible values.

Table 15-9 Possible Values for Tt_state

Value	Description
TT_CREATED	Message has been created but not yet sent. (Only the sender of a message will see a message in this state.)
TT_SENT	Message has been sent but not yet handled.
TT_HANDLED	Message has been handled, return values are valid.
TT_FAILED	Message could not be delivered to a handler.
TT_QUEUED	Message has been queued for later delivery.
TT_STARTED	Attempting to start a process to handle the message.
TT_REJECTED	Message has been rejected by a possible handler. This state is seen only by the rejecting process. The ToolTalk service changes the state back to TT_SENT before delivering the message to another possible handler. If all possible handlers have rejected the message, the ToolTalk service changes the state to TT_FAILED before returning the message to the sender.

Tt_status

A Tt_status code is returned by all functions, sometimes directly and sometimes encoded in an error return value. See Chapter 14, “Handling Errors“, for instructions on how to determine whether the Tt_status code is a warning or an error and for retrieving the error message string for a Tt_status code.

Chapter 14, “Handling Errors“, lists the Tt_status codes. The following information is provided for each status code:

- Message id
- Error message string
- Description
- Solution

The ToolTalk Functional Groupings 16

This chapter describes the ToolTalk functions component of the ToolTalk application programming interface (API). The functions are grouped to perform specific operations; for example, the functions required to initialize the ToolTalk Service. They are grouped under the following headings:

- Initialization Functions
- Message Patterns
- Ptypes
- Sessions
- Files
- Messages
- Objects
- ToolTalk Storage Management
- ToolTalk Error Status
- Exiting
- ToolTalk Error-Handling Macros
- Miscellaneous ToolTalk Functions

See Chapter 17, "ToolTalk Functions/Commands", for a table listing these functions and their descriptions. See the Section 3 man pages for details on these functions.

Initialization Functions

Table 16-1 Initializing and Registering with the ToolTalk Service

Return Type	ToolTalk Function
char *	tt_X_session(const char *xdisplay)
Tt_status	tt_default_session_set(const char *sessid)
char *	tt_open(void)
char *	tt_default_procid(void)
Tt_status	tt_default_procid_set(const char *procid)
Tt_status	tt_ptype_declare(const char *ptid)
Tt_status	tt_ptype_undeclare(const char *ptid)
Tt_status	tt_ptype_exists(const char *ptid)
char *	tt_default_ptype(void)
Tt_status	tt_default_ptype_set(const char *ptid)
int	tt_fd(void)

Message Patterns

Table 16-2 Creating, Filling In, Registering,and Destroying Message Patterns

Return Type	ToolTalk Function
`Tt_pattern`	`tt_pattern_create(void)`
`Tt_status`	`tt_pattern_arg_add(Tt_pattern p, Tt_mode n, const char *vtype, const char *value)`
`Tt_status`	`tt_pattern_barg_add(Tt_pattern m, Tt_mode n, const char *vtype, const unsigned char *value, int len)`
`Tt_status`	`tt_pattern_iarg_add(Tt_pattern m, Tt_mode n, const char *vtype, int value)`

Table 16-2 Creating, Filling In, Registering,and Destroying Message Patterns *(Continued)*

Return Type	ToolTalk Function
Tt_status	tt_pattern_xarg_add(Tt_pattern m, Tt_mode n, const char *vtype, xdrproc_t xdr_proc, void *value)
Tt_status	tt_pattern_address_add(Tt_pattern p, Tt_address d)
Tt_status	tt_pattern_callback_add(Tt_pattern m, Tt_message_callback f)
Tt_category	tt_pattern_category(Tt_pattern p)
Tt_status	tt_pattern_category_set(Tt_pattern p, Tt_category c)
Tt_status	tt_pattern_class_add(Tt_pattern p, Tt_class c)
Tt_status	tt_pattern_bcontext_add(Tt_pattern p, const char *slotname, const unsigned char *value, int length)
Tt_status	tt_pattern_context_add(Tt_pattern p, const char *slotname, const char *value)
Tt_status	tt_pattern_icontext_add(Tt_pattern p, const char *slotname, int value)
Tt_status	tt_pattern_xcontext_add(Tt_pattern p, const char *slotname, xdrproc_t xdr_proc, void *value)
Tt_status	tt_pattern_destroy(Tt_pattern p)
Tt_status	tt_pattern_disposition_add(Tt_pattern p, Tt_disposition r)
Tt_status	tt_pattern_file_add(Tt_pattern p, const char *file)
Tt_status	tt_pattern_object_add(Tt_pattern p, const char *objid)
Tt_status	tt_pattern_op_add(Tt_pattern p, const char *opname)

Table 16-2 Creating, Filling In, Registering,and Destroying Message Patterns *(Continued)*

Return Type	ToolTalk Function
Tt_status	tt_pattern_otype_add(Tt_pattern p, const char *otype)
Tt_status	tt_pattern_scope_add(Tt_pattern p, Tt_scope s)
Tt_status	tt_pattern_sender_add(Tt_pattern p, const char *procid)
Tt_status	tt_pattern_sender_ptype_add(Tt_pattern p, const char *ptid)
Tt_status	tt_pattern_session_add(Tt_pattern p, const char *sessid)
Tt_status	tt_pattern_state_add(Tt_pattern p, Tt_state s)
void *	tt_pattern_user(Tt_pattern p, int key)
Tt_status	tt_pattern_user_set(Tt_pattern p, int key, void *v)
Tt_status	tt_pattern_register(Tt_pattern p)
Tt_status	tt_pattern_unregister(Tt_pattern p)
Tt_status	tt_bcontext_join(const char *slotname, const unsigned *char value, int length)
Tt_status	tt_context_join(const char *slotname, const char *value)
Tt_status	tt_icontext_join(const char *slotname, int value)
Tt_status	tt_xcontext_join(const char *slotname, xdrproc_t xdr_proc, void *value)
Tt_status	tt_bcontext_quit(const char *slotname, const unsigned *char value, int length)

Table 16-2 Creating, Filling In, Registering,and Destroying Message Patterns *(Continued)*

Return Type	ToolTalk Function
`Tt_status`	`tt_xcontext_quit(const char *slotname, xdrproc_t xdr_proc, void *value)`
`Tt_status`	`tt_context_quit(const char *slotname, const char *value)`
`Tt_status`	`tt_icontext_quit(const char *slotname, int value)`

Ptypes

Table 16-3 Declaring, Undeclaring, and Checking Ptypes

Return Type	ToolTalk Function
`Tt_status`	`tt_ptype_declare(const char *ptid)`
`Tt_status`	`tt_ptype_exists(const char *ptid)`
`Tt_status`	`tt_ptype_undeclare(const char *ptid)`

Sessions

Table 16-4 Expressing Interest in Sessions

Return Type	ToolTalk Function
`char *`	`tt_default_session(void)`
`Tt_status`	`tt_default_session_set(const char *sessid)`
`char *`	`tt_initial_session(void)`
`Tt_status`	`tt_session_join(const char *sessid)`
`Tt_status`	`tt_session_quit(const char *sessid)`

Table 16-5 Managing Session Information

Return Type	ToolTalk Function
char *	tt_session_prop(const char *sessid, const char *propname, int i)
Tt_status	tt_session_prop_add(const char *sessid, const char *propname, const char *value)
int	tt_session_prop_count(const char *sessid, const char *propname)
Tt_status	tt_session_prop_set(const char *sessid, const char *propname, const char *value)
Tt_status	tt_session_bprop(const char *sessid, const char *propname, int i, unsigned char **value, int *length)
Tt_status	tt_session_bprop_add(const char *sessid, const char *propname, const unsigned char *value, int length)
Tt_status	tt_session_bprop_set(const char *sessid, const char *propname, const unsigned char *value, int length)
char *	tt_session_propname(const char *sessid, int n)
int	tt_session_propnames_count(const char *sessid)
Tt_status	tt_session_types_load(const char *session, const char *filename)

Files

Table 16-6 Expressing Interest in Files

Return Type	ToolTalk Function
Tt_status	tt_file_join(const char *filepath)
Tt_status	tt_file_quit(const char *filepath)
char *	tt_default_file(void)
Tt_status	tt_default_file_set(const char *docid)

Table 16-7 Managing Files

Return Type	ToolTalk Function
Tt_status	tt_file_move(const char *oldfilepath, const char *newfilepath)
Tt_status	tt_file_copy(const char *oldfilepath, const char *newfilepath)
Tt_status	tt_file_destroy(const char *filepath)

Messages

Table 16-8 Creating Messages

Return Type	ToolTalk Function
Tt_message	tt_onotice_create(const char *objid, const char *op)
Tt_message	tt_orequest_create(const char *objid, const char *op)
Tt_message	tt_pnotice_create(Tt_scope scope, const char *op)
Tt_message	tt_prequest_create(Tt_scope scope, const char *op)
Tt_message	tt_message_create(void)
Tt_message	tt_message_create_super(Tt_message m)

Table 16-9 Filling In Messages and Replies

Return Type	ToolTalk Function
`Tt_status`	`tt_message_address_set(Tt_message m, Tt_address p)`
`Tt_status`	`tt_message_accept(Tt_message m)`
`Tt_status`	`tt_message_arg_add(Tt_message m, Tt_mode n, const char *vtype, const char *value)`
`Tt_status`	`tt_message_arg_bval_set(Tt_message m, int n, unsigned char *value, int len)`
`Tt_status`	`tt_message_arg_ival_set(Tt_message m, int n, int value)`
`Tt_status`	`tt_message_arg_val_set(Tt_message m, int n, const char *value)`
`Tt_status`	`tt_message_arg_xval(Tt_message m, int n, xdrproc_t xdr_proc, void *value)`
`Tt_status`	`tt_message_arg_xval_set(Tt_message m, int n, xdrproc_t xdr_proc, void *value)`
`Tt_status`	`tt_message_barg_add(Tt_message m, Tt_mode n, const char *vtype, const unsigned char *value, int len)`
`Tt_status`	`tt_message_bcontext_set(Tt_message m, const char *slotname, unsigned char *value, int length)`
`Tt_status`	`tt_message_callback_add(Tt_message m, Tt_message_callback f)`
`Tt_status`	`tt_message_class_set(Tt_message m, Tt_class c)`
`int`	`tt_message_contexts_count(Tt_message m)`
`Tt_status`	`tt_message_context_set(Tt_message m, const char *slotname, const char *value)`
`char *`	`tt_message_context_slotname(Tt_message m, int n)`

Table 16-9 Filling In Messages and Replies *(Continued)*

Return Type	ToolTalk Function
Tt_status	tt_message_context_bval(Tt_message m, const char *slotname, unsigned char **value, int *len)
Tt_status	tt_message_context_ival(Tt_message m, const char *slotname, int *value)
char *	tt_message_context_val(Tt_message m, const char *slotname)
Tt_status*	tt_message_context_xval(Tt_message m, const char *slotname, xdrproc_t xdr_proc, void *value)
Tt_status	tt_message_disposition_set(Tt_message m, Tt_disposition r)
Tt_status	tt_message_file_set(Tt_message m, const char *file)
Tt_status	tt_message_handler_ptype_set(Tt_message m, const char *ptid)
Tt_status	tt_message_handler_set(Tt_message m, const char *procid)
Tt_status	tt_message_iarg_add(Tt_message m, Tt_mode n, const char *vtype, int value)
Tt_status	tt_message_icontext_set(Tt_message m, const char *slotname, int value)
Tt_status	tt_message_object_set(Tt_message m, const char *objid)
Tt_status	tt_message_op_set(Tt_message m, const char *opname)
Tt_status	tt_message_otype_set(Tt_message m, const char *otype)
Tt_status	tt_message_scope_set(Tt_message m, Tt_scope s)
Tt_status	tt_message_send_on_exit(Tt_message m)

Table 16-9 Filling In Messages and Replies *(Continued)*

Return Type	ToolTalk Function
Tt_status	tt_message_sender_ptype_set(Tt_message m, const char *ptid)
Tt_status	tt_message_session_set(Tt_message m, const char *sessid)
Tt_status	tt_message_status_set(Tt_message m, int status)
Tt_status	tt_message_status_string_set(Tt_message m, const char *status_str)
Tt_status	tt_message_user_set(Tt_message m, int key, void *v)
Tt_status	tt_message_xarg_add(Tt_message m, Tt_mode n, const char *vtype, xdrproc_t xdr_proc, void *value)
Tt_status	tt_message_xcontext_set(Tt_message m, const char *slotname, xdrproc_t xdr_proc, void *value)
Tt_status	tt_otype_opnum_callback_add(const char *otid,int opnum, Tt_message_callback f)
Tt_status	tt_ptype_opnum_callback_add(const char *ptid, int opnum, Tt_message_callback f)

Table 16-10 Examining Messages

Return Type	ToolTalk Function
Tt_address	tt_message_address(Tt_message m)
Tt_status	tt_message_arg_bval(Tt_message m, int n, unsigned char **value, int *len)
Tt_status	tt_message_arg_ival(Tt_message m, int n, int *value)
Tt_mode	tt_message_arg_mode(Tt_message m, int n)
char *	tt_message_arg_type(Tt_message m, int n)

Table 16-10 Examining Messages *(Continued)*

Return Type	ToolTalk Function
char *	tt_message_arg_val(Tt_message m, int n)
Tt_status	tt_message_arg_xval(Tt_message m, int n, xdrproc_t xdr_proc, void *value)
int	tt_message_args_count(Tt_message m)
Tt_class	tt_message_class(Tt_message m)
Tt_disposition	tt_message_disposition(Tt_message m)
char *	tt_message_file(Tt_message m)
gid_t	tt_message_gid(Tt_message m)
char *	tt_message_handler(Tt_message m)
char *	tt_message_handler_ptype(Tt_message m)
char *	tt_message_id(Tt_message m)
char *	tt_message_object(Tt_message m)
char *	tt_message_op(Tt_message m)
int	tt_message_opnum(Tt_message m)
char *	tt_message_otype(Tt_message m)
Tt_pattern	tt_message_pattern(Tt_message m)
Tt_scope	tt_message_scope(Tt_message m)
char *	tt_message_sender(Tt_message m)
char *	tt_message_sender_ptype(Tt_message m)
char *	tt_message_session(Tt_message m)
Tt_state	tt_message_state(Tt_message m)
int	tt_message_status(Tt_message m)
char *	tt_message_status_string(Tt_message m)
uid_t	tt_message_uid(Tt_message m)
void *	tt_message_user(Tt_message m, int key)
Tt_status	tt_message_send(Tt_message m)
Tt_status	tt_message_destroy(Tt_message m)

Table 16-11 Sending and Destroying Messages

Return Type	ToolTalk Function
`Tt_status`	`tt_message_send(Tt_message m)`
`Tt_status`	`tt_message_destroy(Tt_message m)`

Table 16-12 Receiving, Replying to, Rejecting, and Destroying Messages

Return Type	ToolTalk Function
`Tt_message`	`tt_message_receive(void)`
`Tt_status`	`tt_message_reply(Tt_message m)`
`Tt_status`	`tt_message_reject(Tt_message m)`
`Tt_status`	`tt_message_fail(Tt_message m)`
`int`	`tt_message_status(Tt_message m)`
`Tt_status`	`tt_message_status_set(Tt_message m, int status)`
`char *`	`tt_message_status_string(Tt_message m)`
`Tt_status`	`tt_message_status_string_set(Tt_message m, const char *status_str)`
`Tt_status`	`tt_message_destroy(Tt_message m)`

Objects

Table 16-13 Creating, Moving, and Destroying Objects

Return Type	ToolTalk Function
char *	tt_spec_create(const char *filepath)
Tt_status	tt_spec_prop_add(const char *objid, const char *propname, const char *value)
Tt_status	tt_spec_prop_set(const char *objid, const char *propname, const char *value)
Tt_status	tt_spec_bprop_add(const char *objid, const char *propname, const unsigned char *value, int length)
Tt_status	tt_spec_bprop_set(const char *objid, const char *propname, const unsigned char *value, int length)
Tt_status	tt_spec_type_set(const char *objid, const char *otid)
Tt_status	tt_spec_write(const char *objid)
char *	tt_spec_move(const char *objid, const char *newfilepath)
Tt_status	tt_spec_destroy(const char *objid)

Table 16-14 Using ToolTalk Storage

Return Type	ToolTalk Function
char *	tt_spec_prop(const char *objid, const char *propname, int i)
int	tt_spec_prop_count(const char *objid, const char *propname)
Tt_status	tt_spec_prop_set(const char *objid, const char *propname, const char *value)

Table 16-14 Using ToolTalk Storage *(Continued)*

Return Type	ToolTalk Function
Tt_status	tt_spec_bprop(const char *objid, const char *propname, int i, unsigned char **value, int *length)
char *	tt_spec_propname(const char *objid, int n)
int	tt_spec_propnames_count(const char *objid)
char *	tt_spec_type(const char *objid)
char *	tt_spec_file(const char *objid)
Tt_status	tt_spec_write(const char *objid)
Tt_status	tt_file_objects_query(const char *filepath, Tt_filter_function filter, void *context, void *accumulator)
int	tt_objid_equal(const char *objid1, const char *objid2)
char *	tt_objid_objkey(const char *objid)

Table 16-15 Examining Object Type Information

Return Type	ToolTalk Function
char *	tt_otype_base(const char *otype)
char *	tt_otype_derived(const char *otype, int i)
int	tt_otype_deriveds_count(const char *otype)
Tt_mode	tt_otype_hsig_arg_mode(const char *otype, int sig, int arg)
char *	tt_otype_hsig_arg_type(const char *otype, int sig, int arg)
int	tt_otype_hsig_args_count(const char *otype, int sig)
int	tt_otype_hsig_count(const char *otype)
char *	tt_otype_hsig_op(const char *otype, int sig)

Table 16-15 Examining Object Type Information *(Continued)*

Return Type	ToolTalk Function
`int`	`tt_otype_is_derived(const char *derivedotype, const char *baseotype)`
`Tt_mode`	`tt_otype_osig_arg_mode(const char *otype, int sig, int arg)`
`char *`	`tt_otype_osig_arg_type(const char *otype, int sig, int arg)`
`int`	`tt_otype_osig_args_count(const char *otype, int sig)`
`int`	`tt_otype_osig_count(const char *otype)`
`char *`	`tt_otype_osig_op(const char *otype, int sig)`

ToolTalk Storage Management

Table 16-16 Managing ToolTalk Storage

Return Type	ToolTalk Function
`int`	`tt_mark(void)`
`void`	`tt_release(int mark)`
`void`	`tt_free(caddr_t p)`
`caddr_t`	`tt_malloc(size_t s)`

ToolTalk Error Status

Table 16-17 Retrieving ToolTalk Error Information

Return Type	ToolTalk Function
`Tt_status`	`tt_int_error(int return_val)`
`Tt_status`	`tt_pointer_error(void *pointer)`
`char *`	`tt_status_message(Tt_status ttrc)`

Table 16-18 Encoding Error Values

Return Type	ToolTalk Function
`int`	`tt_error_int(Tt_status ttrc)`
`void *`	`tt_error_pointer(Tt_status ttrc)`

Exiting

Table 16-19 Leaving the ToolTalk Session

Return Type	ToolTalk Function
`Tt_status`	`tt_close(void)`

ToolTalk Error-Handling Macros

Table 16-20 ToolTalk Error-Handling Macros

Return Type	ToolTalk Macro
`int`	`tt_is_err(Tt_status s)`
`Tt_status`	`tt_ptr_error(pointer)`

Miscellaneous ToolTalk Functions

To use these functions, you need to include the ToolTalk header file:

#include <Tt/tt_c.h>

Table 16-21 Miscellaneous ToolTalk Functions

Return Type	ToolTalk Function
`char *`	`tt_file_netfile(const char *filename)`
`char *`	`tt_host_file_netfile(const char *host, const char *filename)`
`char *`	`tt_host_netfile_file(const char *host, const char *netfilename)`
`char *`	`tt_message_print(Tt_message *m)`
`char *`	`tt_netfile_file(const char *netfilename)`
`char *`	`tt_pattern_print(Tt_pattern *p)`

ToolTalk Functions/Commands 17

ToolTalk Functions

This chapter provides a table listing the ToolTalk functions and their descriptions. See the Section 3 man pages for details on these functions

Table 17-1 ToolTalk Functions and Descriptions

Function	Description
`tt_bcontext_join`	Adds the given byte array value to the list of values for the named contexts of all the patterns.
`tt_bcontext_quit`	Removes the given byte-array value from the list of values for the contexts of all patterns.
`tt_close`	Closes the current procid.
`tt_context_join`	Adds the given string value to the list of values for the context of all patterns.
`tt_context_quit`	Removes the given string value to the list of values for the context of all patterns.
`tt_default_file`	Returns the current default file.
`tt_default_file_set`	Sets the default to the specified file.
`tt_default_procid`	Retrieves the current default procid for your process.
`tt_default_procid_set`	Sets the current default procid.

Table 17-1 ToolTalk Functions and Descriptions

Function	Description
`tt_default_ptype`	Retrieves the current default_ptype.
`tt_default_ptype_set`	Sets the default ptype.
`tt_default_session`	Retrieves the current default session identifier.
`tt_default_session_set`	Sets the current default session identifier.
`tt_error_int`	Returns an integer error object that encodes the code.
`tt_fd`	Returns a file descriptor.
`tt_file_copy`	Copies all objects that exist on the specified file to a new row.
`tt_file_destroy`	Removes all objects that exist on the files and directories rooted at *filepath*.
`tt_file_join`	Informs the ToolTalk service that the process is interested in messages that involve the specified file.
`tt_file_move`	Destroys all objects that exist on the files and directories rooted at *newfilepath*, then moves all objects that exist on *oldfilepath* to *newfilepath*.
`tt_file_objects_query`	Instructs the ToolTalk service to find all objects in the named files and pass the objids to the filter function.
`tt_file_quit`	Informs the ToolTalk service that the process is no longer interested in messages that involve the specified file.
`tt_free`	Frees storage from the ToolTalk API allocation stack.
`tt_icontext_join`	Adds the given integer value to the list of values for the contexts of all patterns.
`tt_icontext_quit`	Removes the given integer value to the list of values for the contexts of all patterns.

Table 17-1 ToolTalk Functions and Descriptions

Function	Description
tt_initial_session	Returns the initial session identifier of the ttsession with which the current process indentifier is associated.
tt_int_error	Returns the status of an error object.
tt_is_error	Checks whether status is a warning of an error.
tt_malloc	Allocates storage on the ToolTalk API allocation stack.
tt_mark	Marks a storage position in the ToolTalk API allocation stack.
tt_message_accept	Declares that the process has been initialized and can accept messages.
tt_message_address	Retrieves the address attribute from the specified message.
tt_message_address_set	Sets the address attribute from the specified message.
tt_message_arg_add	Adds a new argument to a message object.
tt_message_arg_bval	Retrieves the byte-array value of the *n*th message argument.
tt_message_arg_bval_set	Sets the byte-array value and the type of the *n*th message argument.
tt_message_arg_ival	Retrieves the integer value of the *n*th message argument.
tt_message_arg_ival_set	Adds an integer value of the *n*th message argument.
tt_message_arg_mode	Returns the mode of the *n*th message argument.
tt_message_arg_type	Retrieves the type of the *n*th message argument.
tt_message_arg_val	Returns a pointer to the value of the *n*th message argument.

Table 17-1 ToolTalk Functions and Descriptions

Function	Description
tt_message_arg_val_set	Changes the value of the *n*th message argument.
tt_message_arg_xval	Retrieves and deserializes the data from a message argument.
tt_message_arg_xval_set	Serializes and sets data into an existing message argument.
tt_message_arg_count	Returns the number of arguments in the message.
tt_message_barg_add	Adds an argument to a pattern that may have a byte-array value that contains imbedded nulls.
tt_message_bcontext_set	Sets the byte-array value of a message's context.
tt_message_callback_add	Registers a callback function to be automatically invoked by `tt_message_receive` whenever a reply or other state-change to this message is returned.
tt_message_class	Retrieves the class attribute from the specified message.
tt_message_class_set	Sets the class attribute for the specified message.
tt_message_context_bval	Retrieves the byte-array value and length of a message's context.
tt_message_context_ival	Retrieves the integer value of a message's context.
tt_message_context_set	Sets the character string value of a message's context.
tt_message_context_slotname	Returns the name of a message's *n*th context.
tt_message_context_val	Retrieves the character string of message's context.
tt_message_context_xval	Retrieves and deserializes the data from a message's context.

Table 17-1 ToolTalk Functions and Descriptions

Function	Description
tt_message_contexts_count	Returns the number of contexts in a message.
tt_message_create	Creates a new message object.
tt_message_create_super	Creates a copy of the specified message and re-addresses the copy of the message to the parent of the specified otype.
tt_message_destroy	Destroys the message.
tt_message_disposition	Retrieves the disposition attribute from the specified message.
tt_message_disposition_set	Sets the disposition attribute for the specified message.
tt_message_fail	Informs the ToolTalk service that the process cannot handle the request just received.
tt_message_file	Retrieves the file attribute from the specified message.
tt_message_file_set	Sets the file attribute for the specified message.
tt_message_gid	Retrieves the group identifier attribute from the specified message.
tt_message_handler	Retrieves the handler identifier attribute from the specified message.
tt_message_handler_ptype	Retrieves the handler ptype attribute from the specified message.
tt_message_handler_ptype_set	Sets the handler ptype attribute for the specified message.
tt_message_iarg_add	Adds a new argument to a message object and sets the value to a given integer.
tt_message_icontext_set	Sets the integer value of a message’s context.
tt_message_id	Retrieves the identifier of the specified message.

Table 17-1 ToolTalk Functions and Descriptions

Function	Description
tt_message_object	Retrieves the object attribute from the specified message.
tt_message_object_set	Sets the object attribute for the specified message.
tt_message_op	Retrieves the operation attribute from the specified message.
tt_message_op_set	Sets the operation attribute for the specified message.
tt_message_opnum	Retrieves the operation attribute from the specified message.
tt_message_otype	Retrieves the object type attribute from the specified message.
tt_message_otype_set	Sets the object type (otype) attribute from the specified message.
tt_message_pattern	Returns the pattern that the specified message matched.
tt_message_receive	Returns a handle for the next message queued to be delivered to the process.
tt_message_reject	Informs the ToolTalk service that the process cannot handle this message.
tt_message_reply	Informs the ToolTalk service that the process has handled the message and filled in all return values.
tt_message_scope	Retrieves the scope attribute from the specified message.
tt_message_scope_set	Sets the scope attribute for the specified message.
tt_message_send	Sends the specified message.
tt_message_send_on_exit	Requests that the ToolTalk service send this message if process exits unexpectedly.

Table 17-1 ToolTalk Functions and Descriptions

Function	Description
`tt_message_sender`	Retrieves the sender attribute from the specified message.
`tt_message_sender_ptype`	Retrieves the sender ptype attribute from the specified message.
`tt_message_sender_ptype_set`	Sets the sender ptype attribute for the specified message.
`tt_message_session`	Retrieves the session attribute from the specified message.
`tt_message_session_set`	Sets the session attribute for the specified message.
`tt_message_state`	Retrieves the state attribute from the specified message.
`tt_message_status`	Retrieves the status attribute from the specified message.
`tt_message_status_set`	Sets the status attribute from the specified message.
`tt_message_status_string`	Retrieves the character string stored with the status attribute for the specified message.
`tt_message_status_string_set`	Sets a character string with the status attribute for the specified message.
`tt_message_uid`	Retrieves the user identifier attribute from the specified message.
`tt_message_user`	Retrieves the user information stored in data cells associated with the specified message object.
`tt_message_user_set`	Stores the user information in data cells associated with the specified message object.
`tt_message_xarg_add`	Adds an argument with an XDR-interpreted value to a message object.
`tt_message_xcontext_set`	Sets the XDR-interpreted byte-array value of a message's context.

Table 17-1 ToolTalk Functions and Descriptions

Function	Description
`tt_objid_equal`	Tests whether two objects are equal.
`tt_objid_objkey`	Returns the unique key of an objid.
`tt_onotice_create`	Creates a message.
`tt_open`	Returns the process identifier for the calling process.
`tt_orequest_create`	Creates a message.
`tt_otype_base`	Returns the base otype of the given otype.
`tt_otype_derived`	Returns the *i*th otype from the given otype.
`tt_otype_deriveds_count`	Returns the number of otypes derived from the given otype.
`tt_otype_hsig_arg_mode`	Returns the mode of the arg'th argument of the sig'th request signature of the given otype.
`tt_otype_hsig_arg_type`	Returns the data type of the arg'th argument of the sig'th request signature of the given otype.
`tt_otype_hsig_count`	Returns the number of request signatures for the given otype.
`tt_otype_hsig_op`	Returns the operation name of the sig'th request signature of the given otype.
`tt_otype_is_derived`	Specifies whether derived otype is derived directly or indirectly from base otype.
`tt_otype_opnum_callback_add`	Automatically returns a callback if the specified opnums are equal.
`tt_otype_osig_arg_mode`	Returns the mode of the arg'th argument of the sig'th notice signature of the given otype.
`tt_otype_osig_arg_type`	Returns the data type of the arg'th argument of the sig'th notice signature of the given otype.

Table 17-1 ToolTalk Functions and Descriptions

Function	Description
tt_otype_osig_args_count	Returns the number of arguments of the sig'th notice signature of the given otype.
tt_otype_osig_count	Returns the number of notice signatures for the given otype.
tt_otype_osig_op	Returns the op name of the sig'th notice signature of the given otype.
tt_pattern_address_add	Adds a value to the address field for the specified pattern.
tt_pattern_arg_add	Adds an argument to a pattern.
tt_pattern_barg_add	Adds an argument with a value that contains imbedded nulls to a pattern.
tt_pattern_bcontext_add	Adds a byte-array value to the values in this pattern's named context.
tt_pattern_callback_add	Registers a callback function that will be automatically invoked by `tt_messsage_receive()` wherever a message matches the pattern.
tt_pattern_category	Returns a category value of the specified pattern.
tt_pattern_category_set	Fills in the category field for the specified pattern.
tt_pattern_class_add	Adds a value to the class information for the specified pattern.
tt_pattern_context_add	Adds a string value to the values of this pattern's context.
tt_pattern_create	Requests a new pattern object.
tt_pattern_destroy	Destroys a pattern object.
tt_pattern_disposition_add	Adds a value to the disposition field for the specified pattern.
tt_pattern_field_add	Adds a value to the file field of the specified pattern.
tt_pattern_iarg_add	Adds a new argument to a pattern and sets the value to a given integer.

Table 17-1 ToolTalk Functions and Descriptions

Function	Description
tt_pattern_icontext_add	Adds an integer value to the values of this pattern's context.
tt_pattern_object_add	Adds a value to the object field of the specified pattern.
tt_patern_op_add	Adds a value to the operation field of the specified pattern.
tt_pattern_opnum_add	Adds an operation number to the specified pattern.
tt_pattern_otype_add	Adds a value to the object type field for the specified pattern.
tt_pattern_register	Registers your pattern with the ToolTalk service.
tt_pattern_scope_add	Adds a value to the scope field for the specified pattern.
tt_pattern_sender_add	Adds a value to the sender field for the specified pattern.
tt_pattern_sender_ptype_add	Adds a value to the sending process's ptype field for the specified pattern.
tt_pattern_session_add	Adds a value to the session field for the specified pattern.
tt_pattern_state_add	Adds a value to the state field for the specified pattern.
tt_pattern_unregister	Unregisters the specified pattern from the ToolTalk service.
tt_pattern_user	Returns a value in the indicated user data cell for the specified pattern object.
tt_pattern_user_set	Stores information in the user data cells associated with the specified pattern object.
tt_pattern_xarg_add	Adds a new argument with an interpreted XDR value to a pattern object.

Table 17-1 ToolTalk Functions and Descriptions

Function	Description
tt_pattern_xcontext_add	Adds an XDR-interpreted byte-array value to the values in this pattern's named context.
tt_notice_create	Creates a message.
tt_pointer_error	Returns the status of specified pointer.
tt_prequest_create	Creates a message.
tt_ptr_error	Returns the status of specified pointer.
tt_ptype_declare	Registers your process type with the ToolTalk service.
tt_ptype_exists	Returns whether indicated ptype is already installed.
tt_ptype_opnum_callback_add	Automatically returns a callback if the specified opnums are equal.
tt_ptype_undeclare	Undeclares the indicated ptype.
tt_release	Frees storage allocated on the ToolTalk API allocation stack.
tt_session_bprop	Retrieves the *i*th value of the named property of the specified session.
tt_session_bprop_add	Adds a new byte-string value to the end of the list of the values for the named property of the specified session.
tt_session_bprop_set	Replaces any current values stored under the named property of the specified session with the given byte-string value.
tt_session_bprop_add	Adds a new byte-string value to the end of the list of values for the named property of the specified session.
tt_session_join	Joins the session named and makes it the default session.
tt_session_prop	Returns the *i*th value of the specified session property.

Table 17-1 ToolTalk Functions and Descriptions

Function	Description
`tt_session_prop_add`	Adds a new character-string value to the end of the list of the values for the property of the specified session.
`tt_session_prop_count`	Returns the number of values stored under the named property of the specified session.
`tt_session_prop_set`	Replaces all current values stored under the named property of the specified session with the given character-string value.
`tt_session_propname`	Returns the *n*th element of the list of currently defined property names for the specified session.
`tt_session_propnames_count`	Returns the number of currently defined property names for the session.
`tt_session_quit`	Informs the ToolTalk service that the process is no longer interested in this ToolTalk session.
`tt_session_types_load`	Merges a compiled ToolTalk types file into the running `ttsession`.
`tt_spec_bprop`	Retrieves the *i*th value of the specified property.
`tt_spec_bprop_add`	Adds a new byte-string value to the end of the list of values associated with the specified spec property.
`tt_spec_bprop_set`	Replaces any current values stored under this spec property with a new byte-string.
`tt_spec_create`	Creates a spec (in memory) for an object.
`tt_spec_destroy`	Destroys an object's spec immediately.
`tt_spec_file`	Retrieves the name of the file that contains the object described by the spec.
`tt_spec_move`	Notifies the ToolTalk service that this object has moved to a different file.

Table 17-1 ToolTalk Functions and Descriptions

Function	Description
`tt_spec_prop`	Retrieves the *i*th value of the property associated with this object spec.
`tt_spec_prop_add`	Adds a new item to the end of the list of values associated with this spec property.
`tt_spec_prop_count`	Returns the number of values listed in this spec property.
`tt_spec_prop_set`	Replaces any values currently stored under this property of the object spec with a new value.
`tt_spec_propname`	Returns the *n*th element of the property name list for this object spec.
`tt_spec_propnames_count`	Returns the number of property names for this object.
`tt_spec_type`	Returns the name of the object type.
`tt_spec_type_set`	Assigns an object type value to the object spec.
`tt_spec_write`	Writes the spec and any associated properties to the ToolTalk database.
`tt_status_message`	Returns a pointer to a message that describes that problem indicated by this status code.
`tt_X_session`	Returns the session associated with the named `X` window system display.
`tt_xcontext_quit`	Removes the given XDR-interpreted byte-array value from the list of values for the contexts of all patterns.

Miscellaneous ToolTalk Functions

To use these miscellaneous ToolTalk functions, you need to include the ToolTalk header file:

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

Table 17-2 Miscellaneous ToolTalk Functions

Function	Description
`tt_file_netfile`	Converts the file specified in `filename` to a `netfilename` that can be passed to other hosts on the network.
`tt_host_file_netfile`	Converts the file specified in `host` to a `netfilename` that can be passed to other hosts on the network.
`tt_host_netfile_file`	Converts the file specified `netfilename` to a path name that is valid on the remote host.
`tt_message_print`	Allows you to print messages that are received but not understood.
`tt_netfile_file`	This function converts the file specified in `netfilename` to a path name that is valid on the local host.
`tt_pattern_print`	Allows you to print out patterns.

Tooltalk Commands

This section provides a table listing the ToolTalk-enhanced operating system commands. See the section 1 man pages for details on these commands.

Table 17-3 ToolTalk Functions and Descriptions

Function	Description
`ttcp`	Copies files and directories in a ToolTalk-safe way.
`ttdbck`	Displays, checks, or repairs ToolTalk data bases.
`rpc.ttdbserverd`	Remove Procedure Call (RPC)-based ToolTalk data base server.
`ttmv`	Move or rename files in a ToolTalk-safe way.
`ttrm, ttrmdir`	Remove files or directories in a Tooltalk-safe way.

Table 17-3 ToolTalk Functions and Descriptions

Function	Description
ttsession	ToolTalk message server.
tt_type_comp	The ToolTalk otype and ptype compiler.
tttar	Archives or de-archives files and ToolTalk objects.

Using ToolTalk Messaging 18

Note – Some code fragments shown in this chapter are taken from a ToolTalk demo program called `tt_toolkit_demo`. See the directory `/usr/dt/examples/tt/tt_toolkit_demo` for the source code, `Makefile` and `README` file.

Telling Your Application about ToolTalk Functionality

Before your application can use 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 application programming interface (API). The Messaging Toolkit provides higher-level functions than the ToolTalk API, 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 following header files:

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

Using the ToolTalk Libraries

The ToolTalk libraries are located in the following directory:

```
/usr/dt/lib
```

The library names are `libttt.a` for the archived libraries and `libttSvc.sl` for the shared libraries.

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 that 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. Your tool, however, 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 to have performed. For example, the following scenario outlines the events a generic editor could expect to perform and to 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, the 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, the user is asked whether to modify the document.)

4. User exits the editor. (If text has unsaved changes, the 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 requests that an editor open a document for editing.
- Each instance of the editor notifies other interested instances of changes it makes to the state of the document.

What Kinds of Operations Do the Tools Perform?

- Each instance of the editor can answer questions about itself and its state, such as "What is your status?"
- Each instance of the editor 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 (read-only).
  - Quitting.

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

- The File Manager must request that the editor open a document for editing.
- An instance of the editor can ask another instance of the editor to save changes to the open document.
- An instance of the editor can ask another instance of the editor 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 the editor 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 the editor 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.

Note – Some of the code fragments used in this section are taken from the `tt_toolkit_demo` source file. This file contains the general commands any application needs to perform that are not specific to any particular application. See the directory `/usr/dt/examples/tt/tt_toolkit_demo` for the source code.

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 are printed out in source format.

To generate a list of the installed ptypes including their signatures:

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

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 executes 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`). See the subdirectories of `/usr/dt/examples/tt` for some example ptypes files.

After you have created a ptype file, you need to install the ptype by running the ToolTalk type compiler. On the command line, type the following:

`% tt_type_comp` *file_name.ptype*

where *file_name.ptype* is the name of the ptype file.

Testing for Existing Ptypes in Current Session

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

```
tt_ptype_exists(const char *ptype_id)
```

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 the following function to merge a compiled ToolTalk type file into the currently running `ttsession`:

```
tt_session_types_load (
          const char *session,
          const char *compile_types_file)
```

where *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.

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 fragment shows how to obtain a procid and how to initialize the toolkit.

```
char *procid = ttdt_open(int *tt_fd,
           const char *ptype_name,
           const char *vendor_name,
           const char *version,
           int send_started)
```

Caution – Your application *must* call `ttdt_open` before any other ToolTalk 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. The `ttdt_session_join` function registers patterns and default callbacks for many standard desktop message interfaces.

```
Tt_pattern *sess_patt = ttdt_session_join(
           const char *session_id,
           Ttdt_contract_cb cb,
           Widget shell,
           void *client_data,
           int join)
```

Note that if an application has ptypes installed, it will normally call the function `ttmedia_ptype_declare` before calling `ttdt_session_join`.

Adding the ToolTalk Service to Event Loop

Your application also needs to add the ToolTalk service to its event loop. If the application is an Xt client, it would use `XtAppAddInput` as follows:

```
XtAppAddInput (app_context,
              tt_fd,
              (XtPointer) XtInputReadMask,
              tttk_Xt_input_handler,
              client_data)
```

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” on page 195, ToolTalk-aware editor applications may also need to perform other tasks, including:

- Declaring a ptype.
- Writing a media load callback.
- Accepting a contract to handle 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.

Declaring a Ptype

If an application has a ptype file that has been installed, the ptypes need to be associated with the application. The convenience function for declaring this association is `ttmedia_ptype_declare`:

```
Tt_status status = ttmedia_ptype_declare(
            char *ptype_name,
            int base_opnum,
            Ttmedia_load_pat_cb cb,
            void *client_data,
            int declare)
```

The callback `cb` will be invoked when the application is asked to service a request supported by the ptype `ptype_name`.

Writing a Media Load Pattern Callback

Before coding an editor application to include any ToolTalk functions, you need to write a media load callback routine. This callback is invoked when another application responds to your application’s `ttmedia_load` call. (See the `tt_toolkit_demo` source code for an example of a media load callback routine.)

Accepting a Contract to Handle a Message

When an application receives a message in its `ttmedia_ptype_declare` handler, it should call the following function to accept a contract to handle the request.

```
Tt_pattern *desktop_patts = ttdt_message_accept (
            Tt_message contract,
            Ttdt_contract_cb cb,
            Widget shell,
            void *client_data,
            int accept,
            int send_status)
```

Replying When Request Is Completed

After your application has completed the operation requested (for example to edit a document), it must reply to the sending application. The following function can be used to do the reply and to return the edited contents of the text to the sender.

```
Tt_message msg = ttmedia_load_reply (
            Tt_message contract,
            const unsigned char *new_contents,
            int new_length,
            int reply_and_destroy)
```

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" on page 197, editor applications can also perform other optional tasks such as tasks that use desktop file interfaces to coordinate with other editors. This section mentions some of the ToolTalk functions you need to include in your editor application so that it can perform these optional tasks.

Requesting Modify, Revert, or Save Operations

The following functions can be used to request modify, revert, or save operations:

- `ttdt_Get_Modified`

- ttdt_Revert
- ttdt_Save

Notifying When a File Is Modified, Reverted, or Saved

The function ttdt_file_event can be used to notify other ToolTalk applications that your application's file has been modified, has reverted, or has been saved.

Quitting a File

The function ttdt_file_quit unregisters interest in ToolTalk events about a file and destroys the associated pattern.

The Messaging Toolkit 19

The ToolTalk Messaging Toolkit is a higher-level interface of the ToolTalk application programming interface (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 on 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 Reference Manual. Table 20-1 lists the functions described in this chapter that partly comprise the ToolTalk Messaging Toolkit.

Table 19-1 ToolTalk Messaging Toolkit Functions

Function	Description
`ttdt_message_accept`	Accepts responsibility for handing a ToolTalk request.
`ttdt_session_join`	Joins a ToolTalk session and registers patterns and default callbacks for many standard desktop messages

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 19-2 ToolTalk Messaging Toolkit Function Fields

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*).
reqArgs	Arguments that must 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.
vtypeargumentName	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>
```

Caution – The tttk functions cache some interned X atoms. This may cause problems if an application closes its X server connection and then opens a new X server connection.

Accepting Desktop Requests

The ttdt_message_accept() function registers in the default session for the following TT_HANDLER-addressed requests:

- Get_Geometry, Set_Geometry, Get_Iconified, Set_Iconified, Get_Mapped, Set_Mapped, Raise, Lower, Get_XInfo
- Pause, Resume
- Quit, Get_Status

See the ttdt_message_accept man page and the man pages for each of these requests.

Registering for Standard Desktop Messages

The ttdt_session_join() function joins a specified session and registers patterns and default callbacks for many standard desktop message interfaces. In particular, ttdt_session_join registers for the following TT_HANDLER-addressed requests:

- Get_Environment, Set_Environment, Get_Locale, Set_Locale, Get_Situation, Set_Situation, Signal, Get_Sysinfo
- Get_Geometry, Set_Geometry, Get_Iconified, Set_Iconified, Get_Mapped, Set_Mapped, Raise, Lower, Get_XInfo
- Pause, Resume, Quit
- Set_Status, Do_Command

See the ttdt_session_join man page and the man pages for each of these requests.

ToolTalk Message Sets 20

Standard message sets help developers to develop applications that will automatically integrate with applications developed by others that follow the same message protocols. Extensive work has been done with leading software suppliers and end-users to define standard message sets. The ToolTalk Standard Message Sets (comprised of the ToolTalk Desktop Services Message Set and the ToolTalk Document and Media Exchange Message Set) are higher-level interfaces of the ToolTalk application programming interface (API) that provide common definitions and conventions to easily achieve control and data integration between applications.

General ToolTalk Message Definitions and Conventions

In the ToolTalk messages there are terms used with specific ToolTalk definitions. This section defines these terms and conventions used in the ToolTalk message man pages.

Table 20-1 Document and Media Exchange Message Set Descriptions

Type of Information	Description
header	A single line that describes the message in the following format: *MsgName*(*Tt_class*) where *MsgName* is the name of the message and *Tt_class* is either Request or Notice.
name	The name of the message and a one-line description of the message.
description	An explanation of the operation (event) that the message requests (announces).

Table 20-1 Document and Media Exchange Message Set Descriptions *(Continued)*

Type of Information	Description
synopsis	A representation of the message in the ToolTalk types-file syntax (similar to the syntax understood by the ToolTalk type compiler `tt_type_comp`) in the following format: <fileAttrib> <opName> (<requiredArgs> [<optionalArgs>]); A synopsis entry is given for each interesting variant of the message. <fileAttrib> - An indication of whether the file attribute of the message can/should be set. <opName> - The name of the operation or event is called the “op name” (or “op”). It is important that different tools not use the same opName to mean different things. Therefore, unless a message is a standard one, its opName should be made unique. A good way to do this is to prefix it with: <Company><Product> e.g., “Acme_Hoarktool_My_Frammistat”. <requiredArgs>, <optionalArgs> - The arguments that must always be included in the message. A particular argument is described in the following format: <mode> <vtype> <argument name> where *mode is one of “in”, ”out”, or “inout”*, *vtype* is a programmer-defined string that describes what kind of data a message argument contains; and *argument name* is the name of the argument. The ToolTalk service uses vtypes to match sent message instances with registered message patterns. By convention, a vtype maps to a single, well-known data type.

Table 20-1 Document and Media Exchange Message Set Descriptions *(Continued)*

Type of Information	Description
required arguments	The arguments that must always be in the message. <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. ToolTalk uses vtypes for the sole purpose of matching sent message instances with registered message patterns. Every vtype should, by convention, map to a single, well-known data type. The data type of a ToolTalk argument is either integer, string, or bytes. The data type of a message or pattern argument is determined by which ToolTalk API function is used to set its value. The argument name is merely a comment hinting to human readers of the semantics of the argument, much like a parameter name in a `C` typedef.
optional arguments	The extra arguments that may be included in a message. Unless otherwise noted, any combination of the optional arguments, in any order, may be appended to the message after the required arguments.
description	An explanation of the operation that the request entreats, or the event that the notice announces.
errors	A list of the error codes that can be set by the handler of the request (or the sender of the notice).

Edict

An *edict* is a notice that looks like a request. If a request returns no data (or if the sender does not care about the returned data), it can sometimes be useful to broadcast that request to a set of tools. Since the message is a notice, no data is returned, no replies are received, and the sender is not told if any tool gets the message.

Handler

The *handler* is the distinguished recipient procid of a request. This procid is responsible for completing the indicated operation.

Notice

A *notice* is a message that announces an event. Zero or more tools may receive a given notice. The sender does not know whether any tools receive its notice. A notice cannot be replied to.

Procid

A *procid* is a principal that can send and receive ToolTalk messages. A procid is an identity (created and handed over by the ToolTalk service on demand (via `tt_open`)), that a process must assume in order to send and receive messages. A single process can use multiple procids; and a single procid can be used by a group of cooperating processes.

Request

A request is a message that asks an operation to be performed. A request has a distinguished recipient, called a handler, who is responsible for completing the indicated operation. A handler may fail, reject, or reply to a request. Any number of handlers may reject a request but ultimately only one handler can fail it or reply to it. If no running handler can be found to accept a request, the ToolTalk service can automatically start a handler. If no willing handler can be found, or if a handler fails the request, then the request is returned to the sender in the 'failed' state.

Errors

A `Tt_status` code can be read from a reply via `tt_message_status`. This status defaults to `TT_OK`, or can be set by the handler via `tt_message_status_set`. In extraordinary circumstances (such as no matching handler) the ToolTalk service itself sets the message status.

In addition to the `Tt_status` values defined by the ToolTalk API, the overview reference page for each set of messages lists the error conditions defined for that set of messages. For each error condition, the overview reference page provides

- Its name
- Its integer value

- A string in the "C" locale that explains the error condition.

Since the ToolTalk Inter-Client Conventions (TICC) are a binary message interface, the integer and string are part of that binary interface; the name is not.

- The integer values of these status codes begin at 1537 (`TT_ERR_APPFIRST + 1`). The first 151 codes correspond to the system error list defined in `intro(2)`.

A standard programming interface for these conventions, which binds the name to the integer value, does not yet exist.

The ToolTalk service allows an arbitrary status string to be included in any reply. Since a standard localized string can be derived for each status code, this status string may be used as a free-form elucidation of the status. For example, if a request is failed with `TT_DESKTOP_EPROTO`, the status string could be set to "`The vtype of argument 2 was 'string'; expected 'integer'`". Handling tools should try to compose the status string in the locale of the requestor. See the `Get_Locale` request.

General ToolTalk Development Guidelines and Conventions

Open protocols are encouraged. A protocol is open largely to the extent that it contains an *anonymous message* (that is, messages that are sent without knowledge of who is to receive them). This section provides guidelines to help you independently develop applications that will successfully interact with any other application that supports the message protocol. These guidelines and principles help ensure that two independently-developed applications will be able to initiate and maintain conventions; and, thus, interact with each other. By following these guidelines, you will enable users of your application to better control and customize their environment.

When you write a ToolTalk application, you need to follow these principles:

1. Always make requests anonymous.
2. Let tools be started as needed.
3. Reply to a request only when the requested operation has been completed.
4. Avoid statefulness whenever possible.
5. Declare one ptype for each role a tool can play.

Always Make Anonymous Requests

To design your application to be completely open, you want the requests to be completely anonymous, that is, the requesting process has no knowledge of which tool instance — or even which tool type — will perform the requested operation. If the requests are sent to a specific process, you unnecessarily restrict how users or potential message recipients can use their resources. If the requests are sent to a specific tool type, you unnecessarily restrict the other kinds of tools that can interact with your tool.

You want your message to describe the operation being requested or the event being reported. You do not want your message to describe the process that should receive the message. The less specific knowledge each tool encodes about the tools with which it will interact, the more flexible the overall system is for the user.

Let Tools Be Started as Needed

To design your protocol to be completely open, you want the system to start tools only as needed. When you let a new tool instance be started only as needed, you provide the user with more flexibility and more efficient use of resources such as CPU, screen real estate, and swap space. The ToolTalk service has several features that assume the responsibility of determining when to start a new tool instance:

- The ToolTalk service allows messages and type signatures to have “start” reliability. Start reliability means that if no eligible recipient of a message is running (or is willing to accept the request), the ToolTalk service will start an instance of the type of tool that is statically registered to handle or observe that message.
- The ToolTalk service allows each process type (*ptype*) to specify the maximum number of its instances that may be started in a given session.
- The ToolTalk service offers each request to all eligible running handlers before it starts a new tool instance. An eligible handler can accept or reject a request based on its own criteria (such as its ability to take on a new task; whether or not it has unsaved changes; idle time; iconic state; or whether or not the user has indicated that the tool is free to accept new work).

Reply When Operation has been Completed

To design your application to be completely open, you want to notify the sending process that its requested operation has been performed; however, the operation invoked by a request sometimes takes a relatively long time to complete compared to the very brief time it takes to send the message. Since the sending process is expecting a reply, your tool can respond in one of two ways:

1. It can reply immediately that it has received the request and then convey the actual results of the completed operation in a later message.
2. It can withhold the reply until the operation has been completed.

We recommend the second policy because ToolTalk messaging is entirely asynchronous: neither a tool (nor the session it is in) is blocked because it has one or more requests outstanding.

Avoid Statefulness Whenever Possible

To design your application to be open, you want each message to make sense by itself whenever possible. When a protocol is stateless, the messages in it avoid dependency on any previous messages or on some state in the assumed recipient.

Declare One Process Type per Role

A ToolTalk protocol is expressed in terms of the *roles* that each tool plays (that is, the kinds of tasks each tool is assigned to perform). A ToolTalk ptype essentially instructs the ToolTalk service how to handle any messages in which a tool is interested that are sent when that tool is not running. To design your protocol to be open, you want to declare one ptype for each role in your protocol. When you declare only one ptype per role in your protocol, you provide users with the flexibility to interchange tools as their needs require. For example, a user may want a sophisticated sound-authoring tool for recording but also prefers a simple audio tool to perform the playback.

Thus, you will sometimes want to include only one message signature per ptype. When you include more than one message signature in the same ptype, you are requiring that *any* program that can handle one message can handle the other messages. For example, a ptype "UWriteIt" can include the two message signatures “Display“ and “Edit” because it is expected that any tool that understands the UWriteIt document format can perform both of these operations.

Developing ToolTalk Applications

Developing ToolTalk-aware applications is a design process. You can enable your application to send and receive ToolTalk messages in a simple three-step process:

1. Determine how your application is to interact with other applications, and with users.
2. Select messages and define their use within the context of your application.
3. Integrate ToolTalk calls and messages into your code.

Note – A demonstration of how you can easily add ToolTalk capability to your existing applications has been integrated with the ToolTalk software product.

♦ **Define how the tools will work together and what operations must be performed.**

A clear understanding of what types of communications your application will require is a critical factor in successful application integration. The best approach to analyzing this issue is to define scenarios that represent how your application will be used. From these scenarios you will be able to determine what interaction needs to take place and what information needs to be exchanged. Detailed scenarios that show exactly what information and status is being passed will greatly help you integrate messaging into your application.

♦ **Select the appropriate messages that accomplish these tasks.**

Once you have determined how your applications will interact with other applications and users, you must determine the specific messages needed to accomplish the required tasks.

First, look at the standard message sets available from industry groups such as ANSI, X3H6, and CFI. Use of these messages is strongly recommended for two reasons:

1. The standard messages provide your application with a well-known and documented interface. This interface allows other developers to independently develop applications that can interface with your work. In addition, it provides an interface around which your customers can build integrated systems.
2. The standard message sets provide your application with the "universal plug-and-play" capability. This capability allows you to provide your customers with the flexibility to use multiple applications to provide a service. By giving your customers a choice of applications to use, they can pick the best tool for a particular job and you are not forced to offer features that you feel your product does not need.

If the standard message sets do not support your design, you will need to develop custom messages.

♦ **Integrate ToolTalk calls and messages into your application.**

Once you have completed the design aspect, you are ready to add the ToolTalk capabilities into your application.

First, you need to include the ToolTalk header file in all files that will use ToolTalk API calls. You will also need to register and initialize the patterns that control the sending and receiving functions.

Next, add the ability to send ToolTalk messages to your code. Based on the knowledge gained from designing the scenarios, it is very straightforward to determine what routines need to send what messages, and what the arguments for each message should be.

Once the ToolTalk service is initialized, your application uses the ToolTalk API calls to create and fill in messages to be sent to other applications.

- If your applications uses a windowing system, you only need to add the calls to activate the ToolTalk service in the event polling loop.
- If your application does not already use a polling loop, you need to create a simple loop that periodically checks for messages.

The ToolTalk Desktop Services Message Set

In order to achieve basic desktop integration, applications need to support a basic set of messages to enable inter-application control. The *ToolTalk Desktop Services Message Set* is the common message set that provides this functionality for all applications. A powerful messaging protocol that benefits both developers and users of desktop applications, the ToolTalk Desktop Services Message Set allows applications to easily interact with other desktop applications. Using the ToolTalk Desktop Services Message Set applications can communicate with each other in a transparent manner, both locally and over networks.

Why the ToolTalk Desktop Services Message Set was Developed

In order to provide integrated control of applications, certain basic features are needed to launch, halt, control display appearance, and pass information regarding input and output data. All applications need to have these facilities so that other applications in the toolset can inter-change basic control information. This kind of functionality enables the development of smart desktops and integrated smart toolsets. Groups of applications can now call on each other to perform tasks and to interact as one solution environment for the end-user.

Key Benefits of the ToolTalk Desktop Services Message Set

The ToolTalk Desktop Services Message Set offers developers two key benefits:

1. Allows basic control of applications without direct intervention from the user. Routine or common procedures may be automated for the convenience of the user.
2. Allows tool specialization through a common set of interactions. All ToolTalk aware applications can perform these functions.

General Description of the ToolTalk Desktop Services Message Set

The ToolTalk Desktop Services Message Set 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. The request and notification messages that comprise the ToolTalk Desktop Services Message Set are listed in Table 20-2.

Table 20-2 The ToolTalk Desktop Services Message Set

Requests	**Notifications**
`Get_Modified`	Modified, Reverted
`Get_Status`	Moved
`Get_Sysinfo`	Saved
`Pause, Resume`	Started, Stopped
`Quit`	Status
`Raise, Lower`	
`Save, Revert`	
`Set_Environment, Get_Environment`	
`Set_Geometry, Get_Geometry`	
`Set_Iconified, Get_Iconified`	
`Set_Locale, Get_Locale`	
`Set_Mapped, Get_Mapped`	
`Set_Situation, Get_Situation`	
`Get_XInfo`	
`Signal`	

Desktop Definitions and Conventions

This section defines terms and error messages unique to the Desktop Services message set. Specific to the desktop services messages are values associated with fields as described in Table 20-3.

Table 20-3 Values Associated with Fields

Field	Associated Value
boolean	A vtype for logical values. The underlying data type of boolean is integer; manipulate arguments of this vtype with `tt_*_arg_ival[_set]()` and `tt_*_iarg_add()`. A zero value means false; a non-zero value means true.
buffer	A volatile, non-shared (for example, in-memory) representation of persistent data.
bufferID	A vtype that uniquely identifies buffers. The underlying data type of bufferID is string. To guarantee bufferID uniqueness, use the form `<internal_counter> <procID>`
messageID	A vtype that uniquely identifies messages. The underlying data type of messageID is string; manipulate arguments of this vtype with `tt_*_arg_val[_set]()` and `tt_*_arg_add()`. To guarantee messageID uniqueness, use the form `<internal_counter> <procID>` `tt_message_id()` returns an opaque string of similar uniqueness. Use `tt_message_id()` to generate a message's messageID; however, the inter-client conventions explicitly include the messageID as a message argument to support inter-operation with other versions of the ToolTalk service.
type	Any vtype that is the name of the kind of objects in a particular persistent-object system. For example, the vtype for the kind of objects in filesystems is File; the vtype for ToolTalk objects is ToolTalk_Object.
vendor *toolName* *toolVersion*	Names of arguments. These strings appear in several of the Desktop Service messages. These strings are not defined rigorously; they are intended to present to the user descriptions of these three attributes of the relevant procid.
view	A screen display, such as a (portion of a) window, that presents to the user part or all of a document.
viewID	A vtype that uniquely identifies views. The underlying data type of viewID is string. To guarantee viewID uniqueness, use the form `<internal_counter> <procID>`

Errors

Table 20-4 describes the Desktop Services error messages; the error messages are listed in order of their message ID.

Table 20-4 Desktop Services Error Messages

Message ID	Error Message	Error Message String	Description
1538	TT_DESKTOP_ENOENT	No such file or directory	
1549	TT_DESKTOP_EACCES	Permission Denied	
1558	TT_DESKTOP_EINVAL	Invalid argument	An argument's value was not valid; for example, a locale in Set_Locale that is not valid on the handler's host. Use this error status only when a more-specific error status does not apply.
1571	TT_DESKTOP_ENOMSG	No message of desired type	A messageID does not refer to any message currently known by the handler.
1610	TT_DESKTOP_EPROTO	Protocol error	A message was not understood because: a. A required argument was omitted. b. An argument has the wrong vtype, or the vtype is not allowed in this message; for example, the vtype boolean in the Get_Geometry message. c. An argument's value is not legal for its vtype; for example, negative values for width in the Set_Geometry message. d. An argument's value is not legal for this message; for example, the PATH=/foo variable in Get_Environment message. In general, this error status indicates that the message is malformed.

Table 20-4 Desktop Services Error Messages *(Continued)*

Message ID	Error Message	Error Message String	Description
1688	TT_DESKTOP_CANCELED	Operation was canceled	The operation was canceled because of direct or indirect user intervention. An example of indirect intervention is the user terminating the handling process, or receipt of a Quit() request. (All messages should be taken as authentically representing the wishes of the user whose uid is indicated by tt_message_uid().)
1689	TT_DESKTOP_ENOTSUP	Operation not supported	The requested operation is not supported by this handler. This error indicates that a handler assumes that, if it rejects a request, no other handler will be able to perform the operation. For example, a request such as Set_Iconified() or a request that refers to a state (such as a bufferID) that is managed by this handler alone. A request failed with this error, distinguishes the case of an incompletely-implemented handler from the case of the absence of a handler. **Note:** Do not use TT_ERR_UNIMP in place of TT_DESKTOP_ENOTSUP as TT_ERR_UNIMP means that a particular feature of ToolTalk itself is not implemented.
1699	TT_DESKTOP_ UNMODIFIED	Operation does not apply to unmodified entities	

Warning – The vtype namespace for persistent objects currently only contains `File` and `ToolTalk_Object`. Vendors who want to define a type should give it a vendor-specific name.

Table 20-5 lists each of the generic messages that constitute the ToolTalk Desktop Services Message Set. For details, see the Section 4 man pages.

Table 20-5 ToolTalk Desktop Services Message Set

Message	Description
`Get_Modified` (*Request*)	Asks whether an entity (for example, a file) has been modified.
`Get_Status` (*Request*)	Requests that a tool's current status be returned.
`Get_Sysinfo` (*Request*)	Retrieves information about a tool's host.
`Modified, Reverted` (*Request*)	Notifies that an entity (for example, a file) has been either modified or reverted to its prior state.
`Pause, Resume` (*Request*)	Requests the specified tool, operation, or data performance to pause or resume.
`Quit` (*Request*)	Requests that an operation, or an entire tool, terminate.
`Raise, Lower` (*Request*)	Raises or lowers a tool's window(s) to the front or back, respectively.
`Save, Revert` (*Request*)	Saves or discards any modifications to an entity (for example, a file).
Saved (*Notice*)	Notifies that an entity (such as a file) has been saved to persistent shortage.
`Set_Environment, Get_Environment` (*Request*)	Requests that a tool's environment either be set or retrieved.
`Set_Geometry, Get_Geometry` (*Request*)	Requests that a tool's on-screen geometry either be set or retrieved.
`Set_Iconified, Get_Iconified` (*Request*)	Requests that a tool's iconic state be set or retrieved.
`Set_Locale, Get_Locale` (*Request*)	Sets or retrieves a tool's locale.

Message	Description
`Set_Mapped`, `Get_Mapped` (*Request*)	Requests that a tool's mapping to the screen be set or retrieved.
`Set_Situation`, `Get_Situation` (*Request*)	Requests that tool's current working directory be set or reported.
`Get_XInfo` (*Request*)	Requests that a tool's X11 attributes be set or retrieved.
`Signal` (*Request*)	Requests that a (POSIX-style) signal be sent to a tool.
`Started`, `Stopped` (*Notice*)	Notifies that a tool has started or terminated.
`Status` (*Notice*)	Notifies that a tool has status information to announce.

The ToolTalk Document and Media Exchange Message Set

The ToolTalk Document and Media Exchange Message Set allows applications to easily share each other's multimedia functionality. Using the ToolTalk Document and Media Exchange Message Set, multimedia applications can communicate with each other in a transparent manner, both locally and over networks, regardless of data formats, compression technology, and other technical issues which has previously confined the use of this technology.

Why the ToolTalk Document and Media Exchange Message Set was Developed

While a few vendors have established inter-operability alliances, the range of possible end-user solutions as been restricted. The ToolTalk Document and Media Exchange Message Set allows any application to transparently share a set of multimedia functions with any other application.

Applications that use these simple protocols can quickly and easily create a ToolTalk interface to an array of multimedia services without concern for a particular service provider. Entire groups of applications can now *plug-and-play* together, integrating sound, video, graphics, telephony, and other media sources into new and exciting applications. 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 will 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. For example, you could create a word processing application that integrates a piece of video into a composition and have the video played by another application.

The ToolTalk Document and Media Exchange Message Set is an efficient set of generic message definitions that provide media control and data exchange. The protocol consists of editor messages for media players, editors, and users.

Key Benefits of the ToolTalk Document and Media Exchange Message Set

The ToolTalk Document and Media Exchange Message Set offers developers two key benefits:

1. Ease of multimedia integration to new and existing software.

   Adding multimedia functionality to any application is now vastly simplified. The ToolTalk Document and Media Exchange Message Set allows you to use other developers' multimedia technologies, thus reducing your development time and expenses while increasing your system functionality.

2. Creates a framework that extends the range of end-user solutions.

   By providing application inter-operability, the ToolTalk Document and Media Exchange Message Set allows end-users and other developers to create new vertical solutions. These solutions, in turn, create new opportunities for your products by opening markets that were previously beyond their scope.

General Description of the ToolTalk Document and Media Exchange Message Set

The ToolTalk Document and Media 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. The ToolTalk Document and Media Exchange Message Set is composed of several request messages, listed in Table 20-6

Table 20-6 ToolTalk Document and Media Exchange Message Set

Requests	Notices
Deposit	*There are no notices in the ToolTalk Document and Media Exchange Message Set.*
Display	
Display, Edit	
Edit	
Print	
Translate	

These messages are oriented towards creating, editing, and using documents of a certain media type. The conventions for this message set allow a container application to compose, display, edit, print, or transform a document of an arbitrary media type without understanding anything about the format of that media type. The ToolTalk service routes container requests to the user's preferred tool for the given media type and operation, including routing the request to an instance of the tool that is already running if that instance is best-positioned to handle the request.

Media Exchange Definitions and Conventions

Media exchange messages are sent and received by tools that display or edit some kind of media. Specific to the media exchange messages are values associated with fields. The parts of a Media Exchange message are defined as follows:

<document>

A vector of bytes with an associated mediaType.

<mediaType>

The name of a media format. The mediaType allows messages about that document to be dispatched to the right editor. Standard mediaTypes include those listed in Table 20-7.

Note – The mediaType list will be extended as required. You can extract a list of the installed mediaTypes from the ToolTalk Types Database.

Table 20-7 Standard Media Types

Name of Format	Description	Company
`ISO_Latin_1`	ISO 8859-1 (+TAB+NEWLINE)	ISO
`EUC`	Multi-National Lang. Supplement	AT&T
`PostScript`	PostScript Lang. Ref. Manual	Adobe
`TIFF`	"TIFF Rev. 5" Technical Memo	Aldus/Microsoft
`GIF`	Graphics Interchange Format	CompuServe
`XPM`	XPM -- The X PixMap Format	Groupe Bull
`JPEG`		ISO/CCITT
`JPEG_Movie`		Parallax
`RFC_822_Message`	RFC 822	NIC
`MIME_Message`	RFC MIME	NIC
`UNIX_Mail_Folder`		
`RTF`	MS Word Technical Reference	Microsoft
`EPS`		
`DT_APPOINTMENT`		HP, Sun, IBM, Novell
`Sun_CM_Appointment`		Sun

abstract mediaType

A family of similar mediaTypes, such as flat text or structured graphics.

vector

A string vtype describing a distance and a direction in a document. The syntax of vectors varies by abstract mediaType.

locator

A string describing a location in a document. The syntax of locators varies by abstract mediaType, but should usually be a superset of vector syntax.

flat text

A family of mediaTypes (such as `ISO_Latin_1`) that consist of a sequence of characters from some character set.

Legal vectors for flat text are:

```
lineVec  ::= Line:[-][0-9]+
charVec  ::= Character:[-][0-9]+
vector  ::= <lineVec>
vector  ::= [<lineVec>,]<charVec>
```

Legal locators for flat text are vectors.

time-based media

A family of media types that consist of time-structured data.

Legal vectors for time-based media include:

```
vector ::= uSeconds:[-][0-9]+
vector ::= Samples:[-][0-9]+
```

Legal locators for time-based media are vectors.

Errors

These definitions are common to all Document and Media Exchange messages. Any differences or additions will be noted in the man pages.

`1700 TT_MEDIA_ERR_SIZE`

The specified size was too big or too small.

`1701 TT_MEDIA_ERR_FORMAT`

The data does not conform to the specified format.

Table 20-8 lists each of the messages that constitute the ToolTalk Document and Media Exchange Message Set. For details see the Section 4 man pages.

Table 20-8 ToolTalk Document and Media Exchange Message Set

Message	**Description**
`Deposit` (*Request*)	Saves the document to its backing store.
`Display` (*Request*)	Displays a document.
`Display`, `Edit` (*Request*)	Loads an X11 selection for display or edit.
`Edit` (*Request*)	Edits or composes a document.
`Print` (*Request*)	Prints a document.
`Translate` (*Request*)	Translates a document from one media type to another media type.

Frequently Asked Questions A

This appendix contains answers to the following questions about the ToolTalk service:

How can a process connect to more than one session
Can you start a ttsession with a known session id
What information does a session id contain
Is there a standard way to announce that a new program has joined a session
Where is my message going
What is the basic flow of a message
What happens when a message arrives to my application
How can I differentiate between messages
Can a process send a request to itself
Can I pass my own data to a function registered by tt_message_callback_add
How can I send arbitrary data in a message
Can I transfer files with the ToolTalk service
How are memory (byte) ordering issues handled by the ToolTalk service
Can I re-use messages
What happens when I destroy a message
Can I have more than one handler per message
Can I run more than one handler of a given ptype
What value is disposition in a message
What are the message status elements
When should I use tt_free
What does the ptype represent
Why are my new types not recognized
Can I declare a ptype that is not in the types database
Is ptype information used if a process of that ptype already exists

Can the ptype definition be modified to always start an instance (whether or not one is already running)
What does tt_ptype_declare do
What is TT_TOKEN
When are my patterns active
Must I register patterns to get replies
How can I observe requests
How do I match to attribute values in static patterns
Why am I unable to wildcard a pattern for TT_HANDLER
Can I set a pattern to watch for any file scoped message
Is file scope in static patterns the same as file_in_session scope
What is the difference between arg_add, barg_add, and iarg_add
What is the type or vtype in a message argument
How do I use contexts
How does ttsession check for matches
How many kinds of scope does the ToolTalk service have
What are the tt_db directories, and what is the difference between the types database and the tt_db directories
What should the tt_db databases contain
What does rpc.ttdbserver do
Do ttsession and rpc.ttdbserver ever communicate
What message bandwidth can be supported
Is there a limit to the message size or the number of arguments
What is the most time efficient method to send a message

What network overhead is involved
Does the ToolTalk service use load balancing to handle requests
What resources are required by a ToolTalk application
What happens if the ttsession exits unexpectedly
What happens if rpc.ttdbserver exits unexpectedly
What happens if a host or a link is down
What does tt_close do
Is message delivery guaranteed on a network
Is there a temporal sequence of message delivery
What is unix, xauth, and des
Can my applications hide messages from each other
Is there protection against interception or imitation
Where are queued messages stored and how secure is the storage
Is the ToolTalk service C2 qualified
How can I trace my message's progress
How can I isolate my debugging tool from all the other tools using the ToolTalk service
Can I use the ToolTalk service with C++
Should I qualify my filenames
Can you tell me about ToolTalk objects
Is there a ToolTalk news group

What is the ToolTalk service?

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.

What files are part of the ToolTalk service?

The ToolTalk files are normally found in the `/usr/dt/bin`, `/usr/dt/lib`, `/usr/dt/include/Tt`, and `man/man1` directories. Table A-1 describes the files.

Table A-1 ToolTalk Files

File Name	Description
`ttsession`	Communicates on the network to deliver messages.
`rpc.ttdbserverd`	Stores and manages ToolTalk object specs and information on files referenced in ToolTalk messages.
`ttcp, ttmv, ttrm, ttrmdir, tttar`	Standard operating system shell commands. These commands inform the ToolTalk service when files that contain ToolTalk objects or files that are the subject of ToolTalk messages are copied, moved, or removed.
`ttdbck`	Database check and recovery tool for the ToolTalk databases.
`tt_type_comp`	Compiles the ptype and otype files, and automatically installs them in the ToolTalk Types database.
`libtt.a, libtt.sl, and tt_c.h, tttk.h`	Application programming interface (API) libraries and header file that contain the ToolTalk functions used by applications to send and receive messages.

Where is the initial X-based ttsession started?

The first call to `tt_open` automatically starts `ttsession` if no `ttsession` is running; however, when using the TriTeal Enterprise Desktop, ttsession is normally started when a user logs in to the desktop.

Where are the ToolTalk type databases stored?

The environment variable TTPATH tells the ToolTalk service where the ToolTalk Types databases reside. The format of this variable is:

```
userDB[:systemDB[:networkDB]]
```

Note – The type files are read in reverse order of TTPATH.

This environment variable also tells the ToolTalk service where to search for database server redirection files. The default locations are listed in Table A-2.

Table A-2 Default Locations of ToolTalk Types Database

Database	Location
user	~/.tt
system	/usr/dt/appconfig/tttypes

Do I need X Windows to use the ToolTalk service?

The ToolTalk service does not use X messages or protocols to deliver messages. The ToolTalk service is only associated with X Windows if you run an X session.

When you run an X session, the session name is advertised as a property (named TT_SESSION) on the root window of the X server. Every process that names that X server as its display gets that X session as its default session. Since the X session is defined to be the group of processes displaying on a particular X display, you do need to run X Windows by definition but *not* because the ToolTalk service requires you to use it.

If there is no X server running at all (for example, you are running a session that consists entirely of character-mode applications running on a dumb terminal), use a *process tree session*. When you run a process tree session, the session name is advertised in the environment variable TT_SESSION. This session is the default session for every process in the tree of processes descending from the process that set the environment variable.

Can I use the ToolTalk service with MIT X?

Yes; however, the SHLIB_PATH must point to /usr/dt/lib for the libtt.sl file.

Where is the session id of the X-session?

To get this identifier, enter the following command:

```
xprop -root | grep TT_SESSION
```

Note – An X session is a session that advertises its session id on the TT_SESSION property of root window.

How does tt_open connect to a ttsession?

After some internal initialization, tt_open tries to find a ttsession.

1. tt_open checks whether the environment variable TT_SESSION is set.

   If this environment variable is set, it uses the value as the id of the ttsession.

   If this environment variable is not set, it checks to see if the DISPLAY environment variable is set.

   - If this environment variable is set, it uses the value as the id of the ttsession.
   - If this environment variable is not set, it checks to see if the TT_SESSION property on root X window (of the machine running the display) is set.

   In the event that none of these environment variables are set, it will start a ttsession itself.

2. tt_open 'pings' the ttsession to make sure it is active.

3. tt_open checks the environment variable TT_TOKEN to determine whether the client was started from a start command for the ptype.

   Once the start ptype is determined, tt-open creates a procid.

4. tt_open creates a TCP/IP socket on the client side to which ttsession connects.

   Activity on the socket is noticed via the socket's associated file descriptor. ttsession only uses this channel to notify the client of incoming messages.

Note – Call tt_close on this file descriptor; do *not* call the close function. If you call the close function on the file descriptors returned by tt_fd, your file descriptor count will rise upon successive tt_open and close calls.

---

5. tt_open refreshes the database hostname redirection map.

▼ After calling tt_open, when does a session actually begin?

If the default session is an X session and there is no ttsession running, libtt starts one; otherwise, the ttsession must be started first in order to get the session name.

▼ If another session is attached, does the first session get killed?

No. The first session will still be running.

▼ How can processes on different machines communicate using the ToolTalk service?

There are two ways in which processes on different machines can communicate using the ToolTalk service.

1. They can connect to the same session.
2. They can scope to a file that is NFS mounted on the machines involved.

Connecting to the Same Session

To connect the processes to the same session, you first need to determine a common interest for the processes (for example, a scheme that associates a session name with the common interest of the processes) and then you need to determine how to propagate the session name to all of the processes. The

ToolTalk service does not provide a mechanism to distribute the session address (other than the possible advertisement of a session id on the TT_SESSION property of the root windows of X servers).

To get a session name, you can use the command

```
ttsession -p
```

which forks off a new session and prints its name to stdout; or you can the command:

```
ttsession -c
```

which sets the environment variable $TT_SESSION to the session id.

You then need to use some mechanism to put that session name in a place where the other processes can find it. Some examples of where you can place the session name are:

- a shared file
- a .plan file
- a mail message
- a separate RPC call of your own design
- NIS

  For example, one approach using a well-known file in a NFS-exported file system can be done as follows:

  a. Start ttsession with the following command:

```
ttsession -p >/home/foo/sessionaddress
```

  b.

  c. Ensure that the clients use the session address from the file; for example, wrap the clients in a shell script which reads the session address and sets TT_SESSION as follows:

```
#!/bin/csh
setenv TT_SESSION ‘cat /home/foo/sessionaddress‘
exec client-program
```

  d.

  Alternately, the processes can use the session name in the tt_default_session_set call to connect to that session.

You could also send messages in the ttsession associated with a particular `X` server to advertise the newly-created ttsessions.

Scoping to a NFS-mounted File

File scoping is when a process registers a file scope pattern. The name of that session is placed on a list in `rpc.ttdbserverd` that is associated with the registered file. When a file-scoped message is sent, the ToolTalk service retrieves the list of sessions for the file and forwards the message to each of the sessions on the `rpc.ttdbserverd` list for that file.

Note – To scope to a file that is NFS-mounted on the machines involved requires a file system to be NFS mounted on all the systems and `rpc.ttdbserverd` to be run on the NFS server.

▼ What is the purpose of tt_default_session_set?

`tt_default_session_set` determines the `ttsession` to which a call to `tt_open` will connect.

▼ How can a process connect to more than one session?

Table A-3 describes several default variables that are used when communicating with the ToolTalk service.

Table A-3 Some Default Variables

Variable	Description
procid	Set by `tt_open`. This variable identifies the client to `ttsession`.
ptype	Set by `tt_ptype_declare`.
file	Set when you join a file. If no file is set in the message, the file attribute is set to the default file.

If you use the API functions for getting and setting the procid, your application can switch between multiple sessions. For example:

```
connect to session 1
store the default procid in filename
connect to session 2,
store the default procid filename
restore associated default procid
interact with particular_session
```

Note – The default file and ptype are part of the current default procid. Changing the default procid also changes the default file and ptype to the default file and ptype associated with that procid.

▼ Can you start a ttsession with a known session id?

No. You have to get the session id from the ToolTalk service.

▼ What information does a session id contain?

The session id consists of a number of fields, including:

- Version of address format
- Unix pid of process
- RPC Transient Program Number
- Unused version (compatibility holdover)
- Authorization level
- User id
- Host IP address
- RPC version

Caution – The format of a session id is a private interface. Do *not* write ToolTalk clients that depend on the format of a session id.

Is there a standard way to announce that a new program has joined a session?

Broadcast a notice message to notify interested processes when a new process joins a session. To observe notice messages, a process that want to be notified if a new process joins a session must register patterns to observe these notices.

Note – The Desktop Services "Started" message was developed for this purpose.

Where is my message going?

Use the `-t` (trace mode) at start-up to observe how ttsession processes each message you send. You can also toggle the trace mode on and off by sending ttsession a USR1 signal; for example:

```
kill -USR1 <ttsession_pid>
```

Alternatively, you can use the `ttsnoop` utility to monitor a message with very general patterns.

What is the basic flow of a message?

Session-Scoped Message Flow

The basic flow of a session-scoped message is as follows:

1. The client builds request message and calls `tt_message_send`.
2. `ttsession` finds a handler.

   The environment variable `TT_TOKEN` is set by `ttsession` when it starts the handler.
3. The handler starts up and calls `tt_open` and `tt_fd` to establish communication to `ttsession`.
4. The handler declares its ptype to `ttsession`.

5. ttsession changes all the static patterns for the ptype into dynamic patterns.

   At this point, the patterns are not active because the handler has not yet joined the session.

6. The handler joins session, activating patterns.

7. ttsession notifies the handler that a message is queued.

8. The handler notices activity on the file descriptor and calls tt_message_receive to retrieve the message.

   If the message returned by tt_message_receive has the status TT_WRN_START_MESSAGE, the ToolTalk service started the process to deliver the message. In this case, messages for the ptype are blocked until the process either replies, rejects, or fails the message (even if it is a notice), or calls tt_message_accept.

9. The handler performs the requested operation.

10. The handler returns a reply to request.

11. ttsession notifies the client that a (reply) message for it is in the queue.

    The client's file descriptor is activated.

Note – The client actually receives a message every time its request message changes state.

---

12. The client calls tt_message_receive to retrieve the result.

File-Scoped Message Flow

The basic flow of a file-scoped message is as follows:

1. A file-scoped pattern is registered.

   libtt notifies the database server about the file and the session in which it is registering the pattern.

2. libtt checks with the database server to find all the sessions that have clients who have registered interest in the specified file.
   - For notices, it communicates with all these sessions directly.

- For requests, it notifies its session about the message and the list of other sessions involved.

3. The sessions communicate amongst each other to find a handler.

What happens when a message arrives to my application?

When a message arrives to your application, the following occurs:

1. The file descriptor becomes active.
2. The Xt main loop breaks out of its select and calls the function registered by the `XtAppAddInput` call.
3. The registered function calls `tt_message_receive`.

   The message is read in and any callbacks associated with the message are run.
4. The message callback returns.
   - If the message callback returns `TT_CALLBACK_PROCESSED`, `tt_message_receive` returns a value of null to the input callback.
   - If the message callback returns `TT_CALLBACK_CONTINUE`, a `Tt_message` handle for the message is returned.
5. The input callback continues with any other processing.

For example, the following input callback

```
input_callback(...)
{
     Tt_message m;
     printf ("input callback entered\n");
     m = tt_message_receive();
     printf ("input callback exiting, message handle is %d\n",
             (int)m);
}
```

and the following message callback

```
message_callback(...)
{
     printf("message callback entered\n");
     return TT_CALLBACK_PROCESSED;
}
```

results in the following output

```
input callback entered
message callback entered
input callback exiting, message handle is 0
```

▼ How can I differentiate between messages?

You can differentiate between messages as follows:

- Each message has an identifier that uniquely identifies the message across all running ttsessions.
- You can use the `tt_message_user` call to include information on a user cell to associate the message to the application's internal state.
- Message handles remain the same. For example, the following code tells you whether the message you received is the same as the message you sent.

```
Tt_message m, n;
m = tt_message_create();
...
tt_message_send(m);

... wait around for tt_fd to become active

n = tt_message_receive();
if (m == n) {
// this is a reply to the message we sent
    if (TT_HANDLED == tt_message_state(m)) {
        // the receiver has handled the message, so we can go on
         ...
     }
} else {
     // this is some new message coming in
}
```

Can a process send a request to itself?

Yes. A process can send a request that gets handled by itself. A typical pattern for this type of request is:

```
{   ...
    tt_message_arg_val_set(m, 1, "answer");
    tt_message_reply(m);
    tt_message_destroy(m);
    return TT_CALLBACK_PROCESSED;
}
```

In the case, however, where the handler and the sender are the same process, the message has already been destroyed when the reply comes back (to the same process). Any messages (such as callbacks or user data) attached to the message by the sender are also destroyed. To avoid this situation, do *not* destroy the message; for example:

```
{   ...
    if (0!=strcmp(tt_message_sender(m),tt_default_procid())) {
        tt_message_destroy(m);
}
```

Can I pass my own data to a function registered by tt_message_callback_add?

To pass your data to a function registered by `tt_message_callback_add`, use the user data cells on the message; for example:

```
        x = tt_message_create();
        tt_message_callback_add(x,my_callback);
        tt_message_user_set(x, 1, (void *)my_data);

....

Tt_callback_action
Tt_message_callback(Tt_message m, Tt_pattern p)
{
    struct my_data_t *my_data;
    my data = (struct my_data_t *)tt_message_user(m, 1);

         ...

}
```

Note – User data can only be seen in the client where the data is sent.

How can I send arbitrary data in a message?

The ToolTalk service does not provide a built-in way to send structs; it only provides a way to send strings, ints, and byte arrays. To send structs, use an XDR routine to turn the struct into a byte array and put the bytes in the message. To deserialize, use the same XDR routine.

Can I transfer files with the ToolTalk service?

No, not directly. You can however:

- Place the file data in a message argument.

  The ToolTalk service copies the message data from the application into the library, from the library to `ttsession`, from `ttsession` to the receiver's library, and then out of the library when the receiver gets the argument value. If the data is large, this method can be very slow and use up a large amount of memory.

- Place the file name in a message argument.

  This method assumes that every receiver mounts the file, and mounts it at the same mount point.

- Place the file name in the `tt_message_file` attribute.

  This method also assumes that every receiver mounts the file; however, the ToolTalk service will resolve any mount point differences.

How are memory (byte) ordering issues handled by the ToolTalk service?

The ToolTalk service allows you to place ints, strings, and byte vectors into messages. An XDR routine ensures that these data types are correct for each client. If you have data that is not one of these three data types, you must serialize the data into a byte vector before you place it into a message.

Can I re-use messages?

No. Messages cannot be sent multiple times with different arguments. They must be iteratively created, sent, and then destroyed.

What happens when I destroy a message?

When you destroy a message, you destroy the handle but *not* the underlying message. The underlying message is destroyed only when ToolTalk is done with it and all the external handles are destroyed. For example, if you destroy a handle to a message immediately after you send it, you will get a new handle when the reply comes back.

Once you destroy a message, however, the ToolTalk service will not show it to you again under any circumstances. For example, if you register a pattern to observe a request you send and then destroy the message when your pattern matches it, you will not see the message when it is in state "handled" (that is, when it is a reply).

Can I have more than one handler per message?

No, not currently. If you want multiple processes, you can use notices; or you can use message rejection to force the ToolTalk service to deliver the request to all the possible handlers — however, each of these handlers must actually perform some kind of operation.

Can I run more than one handler of a given ptype?

Yes, you can run more than one handler of a given ptype; however, the ToolTalk service does not have a concept of load balancing (that is, the ToolTalk service will choose *one* of the handlers and deliver additional matching messages to the chosen handler only). There are several ways to force the ToolTalk service to deliver messages to other handlers:

1. Use `tt_message_reject`.

   If a message comes in and a process does not want to handle it because the process is busy, the process can reject the message. The ToolTalk service will then try the next possible handler (and apply the disposition options when it runs out of registered handlers.)

This method requires the process to be in an event loop; that is, it must call `tt_message_receive` when the `tt_fd` is active; however, if the process is in a heavy computational loop, this method fails.

2. Unregister the pattern when busy. For example:

```
m = tt_message_receive();
if (m is the message that causes us to go busy) {
     tt_pattern_unregister(p);
}
```

The ToolTalk service will not route matching messages to the process when the pattern is not registered. When you want the process to receive messages again, re-register the pattern.

Note – This method causes a race condition. For example, a second message could be sent and routed to the process in the time between the first `tt_message_receive` call and the `tt_pattern_unregister` call.

---

3. A combination of Methods 1 and 2.

You can use a combination of the first two techniques in the following manner:

```
get the message
unregister the pattern
loop, calling tt_message_receive until it returns 0; reject
all the returned messages
handle the message
re-register the pattern
repeat
```

Note – This method assumes that the process only registers one pattern.

▼ What value is disposition in a message?

Message disposition can override the disposition specified in the static type definition. If the message specifies the handler ptype and the message does not match any of the static signatures, the disposition set in the message will be the one followed. For example, if the disposition in the message is `TT_START` and the ptype specifies a start-string, an instance will be started.

What are the message status elements?

The ToolTalk service does not use message_status_string. This message component is for use by the applications. The ToolTalk service only sets the message status if a problem occurs with message delivery; otherwise, this message component is set and read in an application-dependent manner.

When should I use tt_free?

`libtt` maintains an internal storage stack from which you receive data buffers. When you call a ToolTalk API routine, any `char *` or `void *` returned points to a copy that you are responsible for freeing.

Use the mark and release functions to free allocated buffers during a sequence of operations; however, the release call frees *everything* allocated since the corresponding mark call. If you want to store certain data that was returned by the ToolTalk service, make a copy of the data before you do any operations that may free it.

What does the ptype represent?

Ptypes are programmer-defined strings that name tool kinds. (You can roughly translate ptype as *process type*.) Each ptype can be associated with a set of patterns that describe the messages in which that particular ptype is interested and a string for the ToolTalk service to invoke when an instance of that ptype needs to be started.

The main purpose of ptypes is to allow tools to express interest in messages even when no instance of the tool is actually running in the scope in which the message is sent. If a tool is able to perform a message's requested operation, or wants to be notified when a particular message is sent, it indicates this instruction in its ptype and ToolTalk will start the tool when necessary. Since the ptypes database can also be modified by the system administrator or user, the mechanism allows the site's or user's favorite tool be designated as the tool to handle a particular message.

▼ Why are my new types not recognized?

`ttsession` only reads the types database on start-up, or on receipt of a USR1 signal. To recognize new types, you can either restart `ttsession`, or (preferably) force your running `ttsession` to reread the types database by sending it the USR2 signal; for example:

```
kill -USR2 <ttsession pid>
```

▼ Can I declare a ptype that is not in the types database?

Yes. To do this, use the `tt_session_types_load` call.

▼ Is ptype information used if a process of that ptype already exists?

The ToolTalk service always looks for one handler and any number of observers for every message. In this case, even though the ToolTalk service finds a handler running, it will still look through the ptypes for any observe patterns that match the message. If a ptype with an observe pattern that matches does exist and there is no process of that ptype currently running, the ToolTalk service will start a new process or queue the message (as specified in the ptype pattern or in the message).

▼ Can the ptype definition be modified to always start an instance (whether or not one is already running)?

No. Messages to a ptype are blocked during start-up until the ptype either replies to the message, or issues a `tt_message_accept` call; however, the implementation of the ptype can include `tt_message_reject` for any request it gets that do not have a status of `TT_WRN_START_MESSAGE`. All requests will then be delivered to (and rejected by) all running instances of the ptype before a new one gets started. This method will be slow if many of these ptypes are running at the same time, or if the message contains a large amount of data. Alternatively, you could use `tt_message_accept`, which basically unblocks messages to the ptype.

What does tt_ptype_declare do?

When you declare the ptype, your static patterns exist in `ttsession` memory. When a ptype is registered by an application, the ToolTalk service also checks for otypes that mention the ptype and registers the patterns found in these otypes. To activate the static patterns, your application must call the appropriate `join` functions.

Note – Multiple declarations by an application of the same ptype are ignored.

What is TT_TOKEN?

When processing a message that requires an application to be started, the ToolTalk service sets this environment variable in the child process. When the application starts and performs `tt_open`, this information is passed back to the ToolTalk service to inform it that the application coming up is the one started or delegated to handle the message.

When are my patterns active?

A pattern must be registered with the session in which it wants to be active. Patterns can be active for more than one file (for a given procid); the file part of the pattern will match any of the listed files.

Note – Contexts are not scopes. A pattern that is joined to contexts but not joined to any file or session cannot match any message.

Must I register patterns to get replies?

No. You do not need to register patterns to get replies. However, if you do register a pattern that matches a reply, the reply will come through your event loop twice: once because it matched a pattern, and again because it is a reply.

How can I observe requests?

Observers can observe requests *if* the pattern matches *and* the message is *not* point-to-point (that is, `TT_HANDLER`). If your observer pattern is not matching any requests, you can run ttsession in trace mode to find out why.

How do I match to attribute values in static patterns?

The ToolTalk static pattern (that is, types database) mechanism does not allow you to match patterns by attribute values. You can match by file scope or argument vtype but you *cannot* by match by the particular filename or by argument value.

Note – This restriction also applies for matching on contexts in static patterns.

Why am I unable to wildcard a pattern for TT_HANDLER?

You cannot wildcard patterns for `TT_HANDLER`-addressed messages because these messages are not pattern matched.

Can I set a pattern to watch for any file scoped message?

No. Not specifying a file name when you use file scoping is virtually the same as specifying that you want to match to file-scoped messages about every file in the universe.

Note – A session attribute may be set on a file-scoped pattern to emulate file-in-session scoping; however, a `tt_session_join` call will not update the session attribute of a pattern that is scoped as `TT_FILE`.

Is file scope in static patterns the same as file_in_session scope?

No, these scopes have different purposes.

For example, assume all sessions currently have the same static patterns and at least one pattern *P* that will match a message *M* (which you will be sending). No session has any clients that have registered interest in the file `foo.bar`.

You are connected to session *A* and issue a file-scoped message *M* for file `foo.bar`. Since no client of any session has previously expressed any interest in this file, session *A* is the only file that will get the message. (The message will match against static pattern *P* in session *A*.) Once the ptype is started, the pattern actually becomes scoped to file (within that session) and session *A* will honor all the promises.

If all sessions do *not* have the same static patterns, the results are different. For example, session *B* could have an extra pattern *P*' that is file-scoped and that should match message *M*. When message *M* is sent in session *A*, the database server will not send the message to session *B* if no client of session *B* has previously expressed interest in the file `foo.bar`; however, if a client of session *B* has previously expressed interest in the file `foo.bar`, then the database server would know that at least one client in that session was interested in the file `foo.bar` and would send also the message to session *B*.

What is the difference between arg_add, barg_add, and iarg_add?

The `barg_add` and `iarg_add` calls are basically an `arg_add` call followed by a set of the value.

What is the type or vtype in a message argument?

The type or vtype (which is short for *value type*) in a message argument indicates the semantic domain in which the argument's value has meaning and is determined by your application.

Vtypes are analogous to typedefs in C. Every vtype, by convention, corresponds to only one of the three possible data types for argument values.

The vtype mechanism allows you to declare two values as the same type; for example, you could declare both the vtype *messageID* and the vtype *bufferID* as C strings with different semantics for each: some operations are valid on *messageID* only, some operations are valid on *bufferID* only, and some operations are valid on both vtypes. The pattern-matching mechanism makes sure that a request with a *bufferID* string does not get matched to a pattern for an operation that is only valid on *messageID* strings.

How do I use contexts?

You can use contexts to restrict matching. To restrict matching, a message must have the same contexts, or a superset of the contexts, in order to possibly match. Also, if the name of a context slot begins with a dollar sign (`$`) character (for example, `$ISV`) and the message causes an application to be started, the environment variable for the started application will be set to whatever value is indicated in the context slot.

▼ How does ttsession check for matches?

Table A-4 describes the various ways `ttsession` checks for matches.

Table A-4 How ttsession Checks for Matches

Mechanism	Description	Match?
`TT_HANDLER`	This type of addressing is "point-to-point" delivery — the message is passed directly to the receiver. You cannot monitor point-to-point messages because registered patterns are never checked.	No matching required.
`TT_PROCEDURE`	• Scans list of static signatures (sig) that have same operation (op) and collects lists of observers and potential handlers.	=> Match
	• If the sig has no arguments and no contexts	=> No Match
	• If sig prototype (number, type and mode of args) have different values	=> Match
	• If the sig contexts are a subset of the contexts in the message	
	• Saves information for any static observers that require queuing.	
	• Scans through dynamic patterns and adds to lists of observers and potential handlers. To form the lists, ttsession first uses the patterns with operations, then the patterns without operations.	
	• Checks reliability, states, class, address, handler, handler ptype, scope, object, otype, sender, sender_ptype, args, contexts.	
	• Delivers to observers first (because a handler can change state).	
	• Delivers to handler with best match — if more than one handler equally "best" matches, the handler is arbitrarily chosen.	
`TT_OBJECT & TT_OTYPE`	Checks whether the otype argument is filled in	
	• If sig has a different otype	=> No Match
	• If sig has no otype & scope is different	=> No Match
	Otherwise, matches in the same manner as for `TT_PROCEDURE` matching.	

How many kinds of scope does the ToolTalk service have?

Currently, the ToolTalk service has only two kinds of scope: session scope and file scope.

Note – X session is sometimes referred to as a scope; however, the X session is really a session scope.

What are the tt_db directories, and what is the difference between the types database and the tt_db directories?

The ToolTalk types databases store the static ptype and otype definitions. These definitions declare the messages to which applications and objects respond. The ToolTalk types compiler modifies the types database when you add or change static type definitions. Upon starting, ttsession reads in these type files.

The TT_DB database is created by `rpc.ttdbserverd`. The tt_db directories contain the associations between files in this partition and the sessions with patterns interested in these files. It also contains all the object spec information for files in this partition.

What should the tt_db databases contain?

The tt_db databases currently contain the following ten files:

```
access_table.ind
access_table.rec
file_object_map.ind
file_object_map.rec
file_table.ind
file_table.rec
file_table.var
property_table.ind
property_table.rec,
property_table.var
```

The permissions for these files are set to `-rw-r--r--` .

What does rpc.ttdbserver do?

The ToolTalk database server daemon performs three major functional duties:

1. It stores the ToolTalk session ids of sessions with clients that have joined a file using the `tt_file_join` call.
2. It stores file-scoped messages that are queued because the message disposition is `TT_QUEUED` and a handler that can handle the message has not yet been started.
3. It stores ToolTalk objects specs.

Do ttsession and rpc.ttdbserverd ever communicate?

No.

What message bandwidth can be supported?

About 100 small messages per second. Performance mainly depends on how many recipients each message has; that is, notices that do not match any pattern are the cheapest while messages that match many observers are the most expensive.

Is there a limit to the message size or the number of arguments?

No; however, while there is no designed limitation to the size of a ToolTalk message or the number of arguments ToolTalk does copy the data several times (both from one area in the client's address space to another area, and across the RPC connection to and from the server). For example, a megabyte of data in a ToolTalk message would be copied at least 4 times:

- From your storage to the ToolTalk library's storage.
- From the ToolTalk library to the ToolTalk server.
- From the ToolTalk server to the receiver's library.
- From the receiver's library to the final resting place.

If there are processes observing the message, even more copying will take place. In addition, no other messages for this session can be delivered during the copy time because the ttsession process is single-threaded. Therefore, if you plan to send really big chunks of data very often, you probably want to consider using a non-ToolTalk way to pass the data.

▼ What is the most time efficient method to send a message?

Directly to process (that is, addressing the message using `TT_HANDLER`) is faster than procedural messages that match only one receiver.

▼ What network overhead is involved?

The ToolTalk service does *not* use hardware broadcast or multicast. The message is sent directly to the ttsession process for the session (whether across the network or not). When a pattern is registered, it also is sent directly to the ttsession process. The ttsession process matches the message against all the patterns and sends the message directly to only the processes that registered patterns that match the message — if no process on another machine is interested in a message, that machine does not need to wake up and look at it.

▼ Does the ToolTalk service use load balancing to handle requests?

No, the ToolTalk service is not a load-distribution mechanism. If two processes with identical patterns are registered, the ToolTalk service arbitrarily chooses one of the processes and delivers all matched messages to it. You can do load distribution if you unregister the pattern while the process is busy and reject any messages that may have been received before the pattern was unregistered.

▼ What resources are required by a ToolTalk application?

Coarse numbers indicate that several 100K of working set for a sending client, ttsession, and a receiving client is required to process messages. ToolTalk memory requirements do not grow over time.

▼ What happens if the ttsession exits unexpectedly?

When `ttsession` crashes, the `tt_fd` becomes active and most ToolTalk API calls will return the `TT_ERR_NOMP` error message:

```
No Message Passer
```

Most applications assume this message means that something has happened to ttsession and will stop sending or receiving ToolTalk messages. Possible recovery from this situation may include:

- Recognize the `TT_ERR_NOMP` situation.
- Call `tt_close` to clean up the connection from its end.
- Reinitialize the ToolTalk service.
- Call the sequence:

  `tt_open, tt_default_session_join, tt_fd`

- Re-register all patterns and re-declare ptypes.

Note – You may need to manipulate the setting of the environment variable `TT_SESSION` and the value of the `TT_SESSION` property of the root `X` window (if it exists) when you restart a crashed ttsession to take over where the last one left off. Also, you must inform other participants of the crashed session of the restarted session and the new session id so that they can recover.

When ttsession crashes, you will not be able to recover the following:

- Patterns registered by procids in the crashed session.
- Outstanding requests from procids in the crashed session.
- Messages that were passed the `tt_message_send_on_exit` call by procids in the crashed session.
- Session props.
- Session-queued messages.

▼ What happens if rpc.ttdbserver exits unexpectedly?

If `rpc.ttdbserverd` exits unexpectedly, `inetd` will start a new one to replace it. Data may be temporarily unavailable but no data will be lost; however, one or more API calls may return `TT_ERR_DBAVAIL`. If the call returns `TT_OK`, the database server will update the ToolTalk databases appropriately either immediately or when a new database server reads the crash recovery log.

▼ What happens if a host or a link is down?

When TCP notices that a host or a link is down, the TCP connection breaks. When a process connection to `ttsession` breaks, `ttsession` behaves as if the process exited. All the patterns are cleaned up, and the process will receive the error message `TT_ERR_NOMP` if it attempts to send or receive messages.

What does tt_close do?

When you call `tt_close`, `ttsession` only closes the current procid. If the current procid is the last procid to close, it cleans up all the ToolTalk structures created since the `tt_open` call was made. You must call `tt_close` on the file descriptor returned by `tt_fd`; otherwise, your file descriptor count will rise upon successive `tt_open` and `close` calls.

Is message delivery guaranteed on a network?

Yes, delivery is reliable because messages are sent using RPC on TCP/IP.

Is there a temporal sequence of message delivery?

Between a given sender and receiver, message sequence is preserved; that is, if process *A* first sends message *M1* and then later sends message *M2* and both messages are received by process *B*, process *B* will receive message *M1* before it receives message *M2*. There are, however, two special exceptions:

1. If process *B* receives message *M1* and then rejects it, message *M1* is redispatched to process *C*. In the meantime (while process *B* is deciding whether to reply or reject message *M1*), the ToolTalk service continues its message delivery. These subsequent messages can appear to "pass" the first request.

2. If process *B*'s messages are queued, it will receive its queued messages when it declares a ptype that contains the pattern that caused the queuing; however, process *B* may not actually receive its queued messages (in this case, message *M1*) until it has already received subsequent messages from process *A*.

What is unix, xauth, and des?

These are the three kinds of authentication:
- *unix* tells you the uid of the entity that is making an rpc call on you. The dbserver enforces security on each RPC call and uses this kind of authentication by default.
- *xauth* uses a read-protected file in your home directory to control access to your `X` display (and, thus, to your ttsession).
- *des* uses the Data Encryption Standard (DES) to ensure that processes that talk to `ttsession` are really who they say they are.

Can my applications hide messages from each other?

No. The ToolTalk service intentionally does not provide a mechanism that allows one application to lock out other applications from seeing its messages.

Is there protection against interception or imitation?

No. The “plug-and-play” concept of the ToolTalk service allows applications to install and deinstall tools of choice that best perform a particular task. If application *B* responds better to protocol `X` than does application *A*, protocol `X` should be allowed to deinstall application *A* and install application *B*.

Where are queued messages stored and how secure is the storage?

File-scoped queued messages are stored in a database on the same filesystem as the file to which they are scoped. The database is readable to the super-user only, and the ToolTalk database server (running as root) only gives the messages to processes owned by a user with read access on the file.

Session-scoped queued messages are stored in the address space of the ttsession that manages the session. ttsession only gives the messages to a process that has satisfied the authentication mode in which the ttsession is running.

Is the ToolTalk service C2 qualified?

No.

How can I trace my message’s progress?

To trace your message’s progress, turn on the trace output of the ttsession involved. Enter the following command:

```
kill -USR1 <unix_pid_of_the_ttsession_process>
```

How can I isolate my debugging tool from all the other tools using the ToolTalk service?

To isolate your debugging tool, use the "process tree session" mode. This mode places the session name in an environment variable to find the `ttsession` process. To use this mode, do the following:

1. Start a new process tree session with trace mode turned on.

```
% ttsession -t -c $SHELL
*
* ttsession (version 1.0, library 1.0)
*
ttsession: starting
%
```

   ttsession starts, sets the environment variable, and forks the given command ($SHELL). You are now running in a subshell. All the commands run from this subshell will use the ttsession started from the command line. You can check the value of the TT_SESSION environment variable for the session id of this new ttsession.

2. Inside the subshell, run the test programs:

```
% ./my_receiver &
[1] 4532
% ./my_sender &

.. and look at the output of the ttsession trace.
```

3. Exit the subshell after testing.

   If you start any tool that uses the ToolTalk service in the subshell, it uses the process tree ttsession, not the X-session ttsession, which will produce undefined results.

▼ Can I use the ToolTalk service with C++?

Yes. The ToolTalk API header file is set up to deal with C++. When you use C++, tt_c.h declares all the API calls as extern C.

▼ Should I qualify my filenames?

No. The ToolTalk service does not allow explicit hostname qualification of pathnames. If you use a filename that contains a colon (:) symbol, the ToolTalk service searches for a filename that contains the colon symbol. The tt_message_file and tt_default_file calls return the *realpath* of the specified file as it appears on the machine on which you invoked the call. The ToolTalk service ensures that

a. If two clients file-scope to the same file on different machines, they can talk to each other without regard to how the two files are actually mounted on each machine.

b. A locally-valid, canonical pathname is returned back to you.

▼ Can you tell me about ToolTalk objects?

ToolTalk objects are somewhat different from what you normally encounter in typical object-oriented languages.

Otypes and inheritance are for implementation only. Two specs can be of the same otype but have different properties — they only share the operations as defined by the signatures in the otype declaration. For each signature in the otype declaration, a ptype must be designated. The designated ptype (process-type) is the 'execution engine' for this operation on an object of this otype. The file part of a spec is similar to a required property: every spec must have a file name; however, that file does not need to exist. The filename part of the spec performs several functions, including:

1. Allows you to specify the host and partition on which the spec will be stored.
2. Provides a grouping mechanism for objects.
3. Allows the ToolTalk-enhanced standard operating commands (such as the `ttmv` command) to keep the database's view of the world consistent with the real world.

▼ Is there a ToolTalk news group?

Yes. The ToolTalk news group is *alt.soft-sys.tooltalk*.

Glossary

Category

Attributes of a pattern that indicate whether the application wants to handle requests that match the pattern or only observe the requests.

contexts

Associates arbitrary pairs (that is, <name. value> pairs) with ToolTalk messages and patterns.

dynamic message patterns

Provides message pattern information while your application is running.

fail a request

Inform a sending application that the requested operation cannot be performed.

fd

File descriptor.

file

A container for data that is of interest to applications.

libtt

The ToolTalk application programming interface (API) library.

handle a message

To perform the operation requested by the sending application; to send a ToolTalk reply to a request.

initial session

The ToolTalk session in which the application was started.

mark

An integer that represents a location on the API stack.

message

A structure that the ToolTalk service delivers to processes. A ToolTalk message consists of an operation name, a vector of type arguments, a status value or string pair, and ancillary addressing information.

message callback

A client function. The ToolTalk service invokes this function to report information about the specified message back to the sending application; for example, the message failed or the message caused a tool to start.

message pattern

Defines the information your application wants to receive.

message protocol

A message protocol is a set of ToolTalk messages that describe operations the applications agree to perform.

notice

A notice is informational, a way for an application to announce an event.

object content

Object content is managed by the application that creates or manages the object and is typically a piece, or pieces, of an ordinary file: a paragraph, a source code function, or a range of spreadsheet cells.

object files

Files that contain object information. Applications can query for objects in a file and perform operations on batches of objects.

object-oriented messages

Messages addressed to objects managed by applications.

object specification (spec)

An object specification (known as a spec) contains standard properties such as the type of object, the name of the file in which the object contents are located, and the object owner.

object type (otype)

The object type (otype) for your application provides addressing information that the ToolTalk service uses when delivering object-oriented messages.

object type identifier (otid)

Identifies the object type.

observe a message

To only view a message without performing any operation that may be requested.

observe promise

Guarantees that the ToolTalk service will deliver a copy of each matching message to ptypes with an observer signature of start or queue disposition. The ToolTalk service will deliver the message either to a running instance of the ptype, by starting an instance, or by queueing the message for the ptype.

opaque pointer

A value that has meaning only when passed through a particular interface.

package

A group of components that together create some software. A package contains the executables that comprise the software, but also includes information files and scripts. Software is installed in the form of packages.

pattern callback

A client function. The ToolTalk service invokes this function when a message is received that matches the specified pattern.

process

One execution of an application, tool, or program that uses the ToolTalk service.

process-oriented messages

Messages addressed to processes.

procid

The process identifier.

ptid

The process type identifier.

ptype

The process type.

reject a request

Tells the ToolTalk service that the receiving application is unable to perform the requested operation and that the message should be given to another tool.

request

A request is a call for an action. The results of the action are recorded in the message, and the message is returned to the sender as a reply.

rpc.ttdbserverd

The ToolTalk database server process.

scope

The attribute of a message or pattern that determines how widely the ToolTalk service looks for matching messages or patterns.

sessid

Identifies the session.

session

A group of processes that are related either by the same desktop or the same process tree.

signatures

A pattern in a ptype or otype. A signature can contain values for disposition and operation numbers.

- Ptype signatures *(psignatures*) describe the procedural messages that the program wants to receive.
- Otype signatures (*osignatures*) define the messages that can be addressed to objects of the type.

spec

See object specification.

static message patterns

Provides an easy way to specify the message pattern information if you want to receive a defined set of messages.

tool manager

A program used to coordinate the development tools in the environment.

ToolTalk Types Database

The database that stores ToolTalk type information.

ttdbck

Check and repair utility for the ToolTalk database.

ttsession

The ToolTalk communication process.

tt_type_comp

The ToolTalk type compiler.

wrapped shell commands

ToolTalk-enhanced shell commands. These commands safely perform common file operations on ToolTalk files.

xdr format tables

The types database read when `ttsession` is invoked.

Index

D

O

P

T

U

V

W

X