ToolTalk™ Programmer's Guide

Document Number 007-1611-020

ToolTalk™ Programmer's Guide
Document Number 007-1611-020

Contents

Figures

Tables

Introduction

This manual describes the ToolTalk™ 1.0 service, an inter-application message service, and how you modify your application to send and receive ToolTalk messages. After you have read this manual you should have an understanding of:

- What the ToolTalk service is and how it works
- What is required to integrate with the ToolTalk service
- How to modify your application to send messages addressed to processes or ToolTalk objects
- How to register message pattern information for the messages you want to receive
- How to receive and handle messages delivered to your application by the ToolTalk service
- How to create and manage ToolTalk objects in your application's data

This manual is intended for users running IRIX 5.2 or later. This manual does not provide separate ToolTalk installation procedures. To install the ToolTalk service, see the *ToolTalk Setup and Administration Guide*.

Who Should Use This Book

This manual is for application developers who create or maintain applications and wish to use the ToolTalk message service. This manual assumes familiarity with the IRIX operating environment.

How This Book Is Organized

This manual is organized as follows:

Chapter 1, "ToolTalk Overview," describes how the ToolTalk service works and how it uses information that your application supplies to deliver messages. This chapter also lists the files installed in the ToolTalk product.

Chapter 2, "Participating in ToolTalk Sessions," describes the location of the ToolTalk 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 3, "Dynamic Message Patterns," describes dynamic message pattern attributes. This chapter also 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 4, "Static Message Patterns," describes static message pattern attributes. This chapter also describes how to provide process and object type information at installation time; how to make a static message pattern available to the ToolTalk Service; how to declare a ptype.

Chapter 5, "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 a callback to requests that will automatically call your callback routine when the reply to your request is delivered to your application.

Chapter 6, "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 7, "Objects," describes how to create ToolTalk specification objects for the objects your process creates and manages.

Chapter 8, "Managing Information Storage," describes how to manage and remove objects.

Chapter 9, “Handling Errors,” describes how to handle error conditions.

Chapter 10, “ToolTalk API,” describes each of the ToolTalk functions and the ToolTalk enumerated types.

Appendix A, “Quick Reference to ToolTalk API,” describes the ToolTalk functions in alphabetical order.

Appendix B, “ToolTalk API Summary (Functional Grouping),” describes the ToolTalk functions by the task to be performed such as registering with the ToolTalk service, sending a message, or creating a message pattern.

Appendix C, “Initialization Error Messages,” describes the ToolTalk errors that may occur during initialization or startup.

Appendix D, “ToolTalk Error Messages,” describes the ToolTalk error messages found in the message catalog.

Related Books

The *ToolTalk Setup and Administration Guide* is aimed at system administrators. It tells them how to set up the ToolTalk service and maintain its files.

Typographic Conventions and Symbols Used in this Manual

Table In-1 describes the typographical conventions and symbols used in this manual.

Table In-1 Typographical Conventions

Typeface	Description	Example
Monospaced	The names of commands; also on-screen computer output; also code samples	Use `ls -a` to list all files. `system% You have mail.`
Boldface monospaced	What you type, contrasted with on-screen computer output	`system%` **`su`** `password:`
Italics	Book titles, new words or terms, file and directory names, variables (to be replaced by a real name or value), and words to be emphasized	Read Chapter 6 in *User's Manual*. These are called *class* options. Edit your *.login* file. To delete a file, type `rm` *filename*. You *must* be root to do this.
The following characters, when they appear in command-line examples, are prompts indicating attributes of the shell to be used:		
%	UNIX C shell prompt	`system%`
$	UNIX Bourne shell prompt	`system$`
#	Superuser prompt, either shell	`system#`

Chapter 1

ToolTalk Overview

The ToolTalk service allows an independent application to communicate with other applications without having direct knowledge of the other applications. To communicate, applications create and send ToolTalk messages. The ToolTalk service receives these messages, determines the recipients, and then delivers the messages to the appropriate applications (see Figure 1-1).

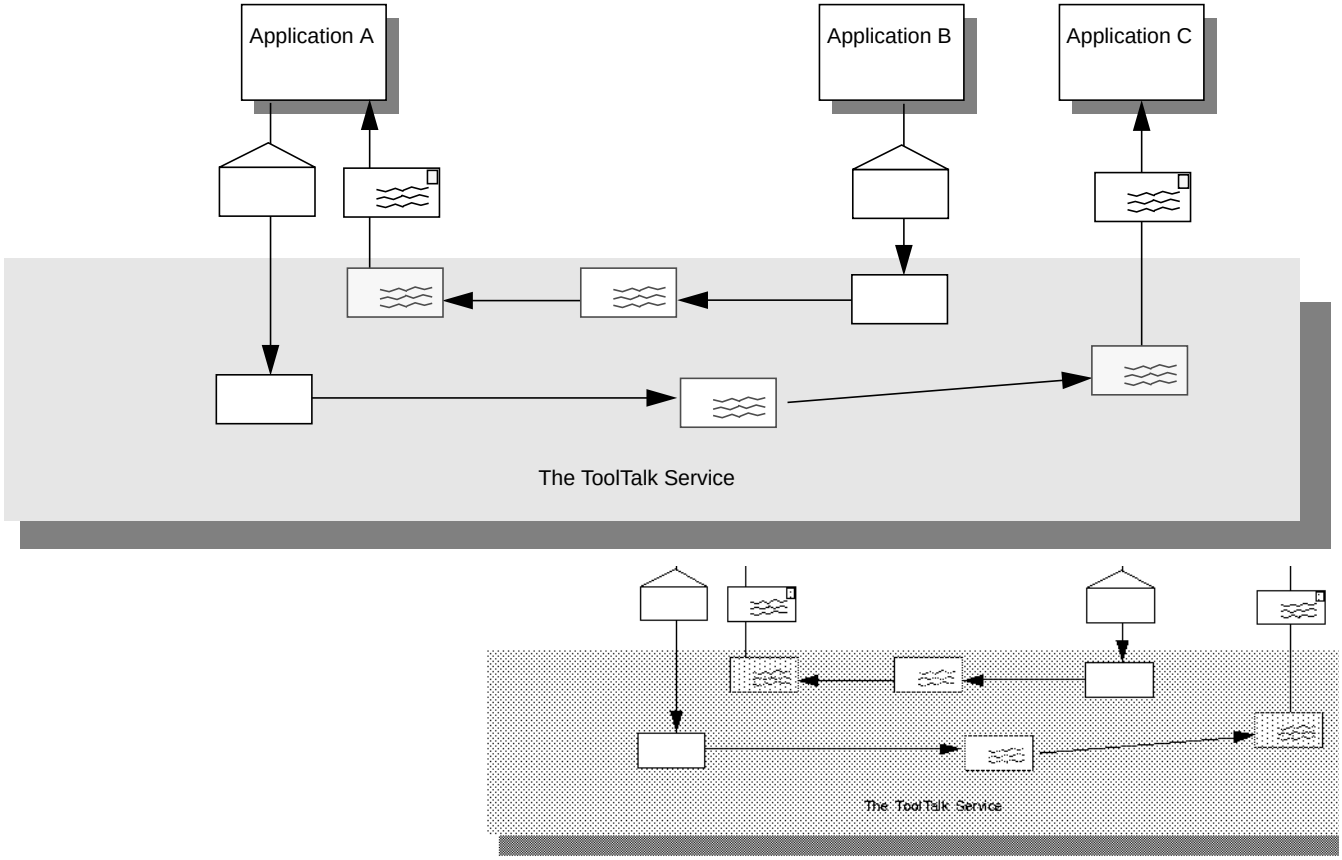


Figure 1-1 Applications Using the ToolTalk Service

ToolTalk Scenarios

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

Connecting and Coordinating Programs

In computer-aided software engineering (CASE), the ToolTalk service provides a way to connect and coordinate individual programs in a programming environment.

In this scenario, Justine uses the following tools in her ToolTalk-based developer's environment:

- a tool manager (a program to coordinate the development tools in the environment)
- a graphical debugger
- a call grapher
- an editor
- a source browser

These tools have been modified to use the ToolTalk service and implement the messages described in Table 1-1.

Message	Description
Started	Informs tool manager that this tool is started
Stopped	Informs tool manager that this tool is stopped
Launch	Requests a certain tool to start
Quit	Requests a certain tool to stop
Display	Requests an edit tool to load and scroll the file to a particular line number
GetSelection	Requests that the tool with the current selection return the file name and line number

Table 1-1 CASE Message Protocol

To determine the cause of a particular error message,

1. Justine starts the tool manager.

   From the tool manager, she double-clicks on the source browser and graphical debugger icons to start them.

The tool manager sends a Launch message to each tool.

As the tools start, they send a Started message to the tool manager with initialization information.

2. Justine loads a source code file in the source browser.

   She finds where the error message is located in the source code and selects the text.

   She selects the Set BreakPoint menu item on the graphical debugger tool.

3. The graphical debugger sends a GetSelection message to the tools currently running in the environment.

   The source browser returns the file name and line number of the selected text.

4. The graphical debugger loads the file, moves to the specified line number, and sets the breakpoint.

5. Justine runs the program and locates the call which results in the error message.

   She selects the Show Call Graph menu item on the graphical debugger tool.

6. The graphical debugger sends a CallGraphFunction message.

   The ToolTalk service starts the installed call grapher and delivers the message.

7. The call grapher loads the call graph for the specified file and scrolls to the specified function.

8. Justine sees another suspicious function that is called just before the function that is producing the error.

   She double-clicks on the suspicious function and the call grapher sends a Display message.

   The ToolTalk service starts an editor and delivers the Display message.

9. The editor loads the file and scrolls to the specified line number.

10. Justine corrects the error and successfully runs her program.

11. From the tool manager, she shuts down all tools.

Automating the Design Process

In the computer-aided design (CAD) of hardware components, tools that are able to communicate with each other help automate the design process for the hardware engineer.

In this scenario, Dustin uses a tool control program that orchestrates tool sequences and CAD tools that have been modified to use the ToolTalk service. All tools use a CAD message protocol that includes the messages described in Table 1-2.

Message	Description
ToolStarted	Informs interested tools that this tool has started
ToolFinished	Informs interested tools that this tool has stopped
DesignOpened	Informs interested tools that a particular design data set has been opened for access
DesignWrite	Requests that a certain tool begins to write to a particular design data set
DesignWriteDone	Informs interested tools that a particular design-write operation has been completed
DesignRead	Requests that a certain tool begin to read a particular design data set
DesignReadDone	Informs interested tools that a particular design-read operation has been completed

Table 1-2 CAD Message Protocol

1. Dustin starts the tool control program.

   As a new tool is needed in the design sequence, he starts the required tool from the control program.

   Each tool initializes with the ToolTalk service and sends out a ToolStarted message to notify the control program that it is now running.

2. Dustin loads a design into a PC layout tool for editing purposes.

When the tool has loaded the design, it sends out a DesignedOpened message, which notifies other tools in the environment that it has opened the file and begun to write design data.

3. When Dustin finishes editing the data, the layout tool sends a DesignWriteDone message, which signals the control program that the edit is complete.
4. The control program then sends a DesignRead message to the next tool required in the design sequence.

How Applications Use ToolTalk Messages

Applications create, send, and receive ToolTalk messages to communicate with other applications. *Sending applications* create, fill in, and send a message; the ToolTalk service determines the recipients and delivers the message to the *receiving applications*. Receiving applications 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 sending applications need to know little about the receiving applications because applications that want to receive messages explicitly state how these messages should appear. 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 registed 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 message, 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.

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 group of processes running in the same X session or process tree session is called a *session* in this manual. A session also contains an instance of the ToolTalk communication program, `ttsession`.

When a process opens communication with the ToolTalk service, the session in which the action takes place becomes the default session for the process. The procid for the process in this case is provided by `ttsession`.

The concept of a session is important in the delivery of messages. Sending applications 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 manual.

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

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

Modifying Applications to Use the ToolTalk Service

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

To use the ToolTalk service, an application calls ToolTalk functions from the ToolTalk 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 and start a session with the ToolTalk service.
- Register message patterns with the ToolTalk service.
- Send and receive messages.
- Unregister message patterns and close your ToolTalk session.

ToolTalk Architecture

The following ToolTalk service components work together to provide interapplication communication and object information management:

- `ttsession` is the ToolTalk communication process.
- One `ttsession` runs in an *X* server session or process tree session and communicates with other `ttsession` processes 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 which 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.

The ToolTalk service uses the Remote Procedure Call (RPC) to communicate between these ToolTalk components.

Applications provide the ToolTalk service with process and object type information in a type database in XDR format.

Figure 1-2 illustrates the ToolTalk service architecture.

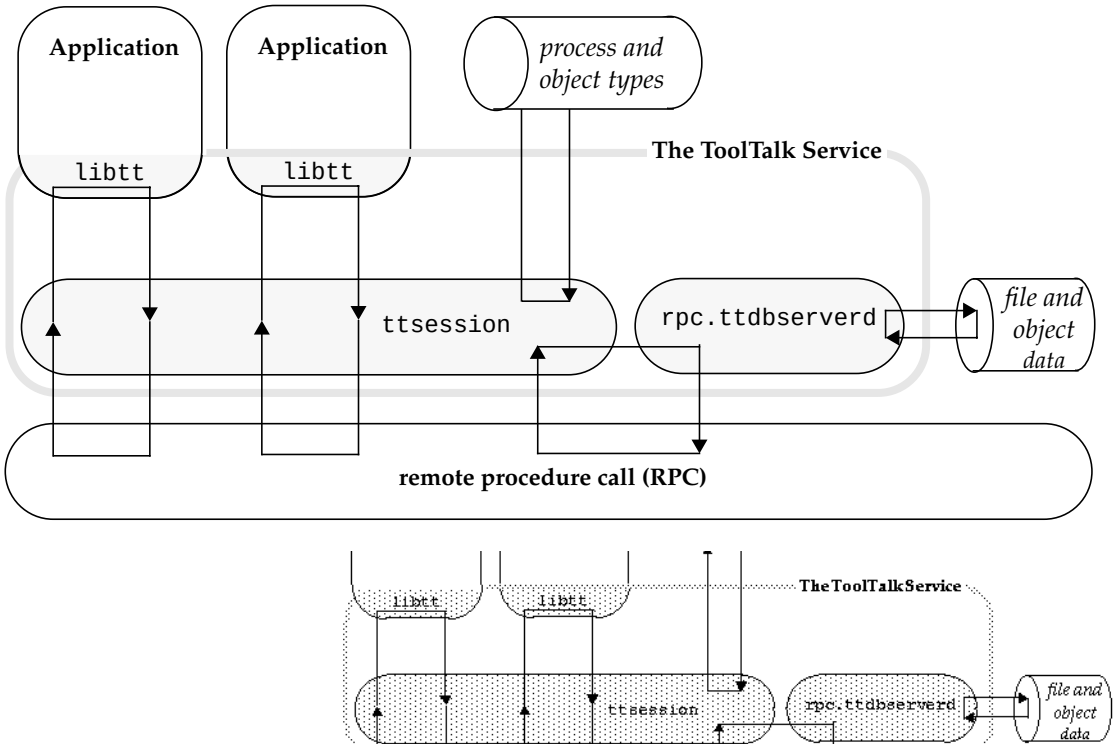


Figure 1-2 ToolTalk Service Architecture

Starting a ToolTalk Session

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

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

```
ttsession[-A max_active_msgs][-a level][-d
display][-s][-t][-v][-h][-c command]
```

See Table 1-3 for a description of the ttsession parameters. (NOTE: If neither the -c or -d parameter is specified, the server specified in the $DISPLAY environment variable is used to start the X session.)

`ttsession` responds to two signals.

- If it receives the `USR1` signal, it toggles the trace mode on or off.
- If it receives the `USR2` signal, it rereads the types file.

Argument	Description
-A *max_active_msgs*	Specifies the maximum number of messages in-progress before a TT_ERR_OVERFLOW condition is returned. Default is 2000.
-a *level*	Sets the server authentication level. The level must be *unix* or *xauth*.
-d *display*	Directs ttsession to start an X session for the given display. Normally, ttsession uses the $DISPLAY environment variable.
-s	Enables silent operation; warning messages are not printed.

Table 1-3 ttsession Parameters

Argument	Description
-t	Turns on trace mode. If trace mode is turned on while ttsession is running, messages appear on the console. Use this mode to see how messages are dispatched and delivered. Trace mode displays the state of a message when it is first seen by `ttsession`. It also displays any attempt to send the message to a given process with the success of that attempt. To toggle the trace mode on or off, use the `USR1` signal.
-v	Prints the version number and exits.
-h	Prints help on how to invoke `ttsession` and exits.
-c *command*	Starts a process tree session and runs the given command. The special environment variable `_SUN_TT_SESSION` will be set to the name of this session. Any process that was started with this environment variable will default to be in this session. This 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. If *command* is omitted, the value of `$SHELL` is used instead.

Table 1-3 ttsession Parameters

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
```

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 the atom `_SUN_TT_SESSION` on the root of screen 0; the host and port number is given for communication with the process managing the session.

Location of the ToolTalk Service Files

The ToolTalk directories and files are described in the following table.

Directory and File(s)	Description
/usr/ToolTalk/bin/ *ttsession*	*/usr/ToolTalk/bin/ttsession* is linked from */usr/sbin/ttsession.* ttsession is the ToolTalk communication process.
/usr/ToolTalk/bin/ *ttcopy* *ttmv* *ttrm* *ttrmdir* *tttar*	The */usr/ToolTalk/bin/tt* files* are linked from the corresponding *t** symbolic links. These files are shell commands that have been enhanced to inform the ToolTalk service when files that contain ToolTalk objects or files that are the subject of ToolTalk messages are copied, moved, or removed.
/usr/ToolTalk/etc *rpc.ttdbserverd*	*/usr/ToolTalk/etc/rpc.ttdbserverd* is linked from *usr/etc/rpc.ttdbserverd*. rpc.tdbserverd stores and manages ToolTalk object specifications and information on files referenced in ToolTalk messages.
/usr/ToolTalk/bin/ *ttdbck*	*/usr/ToolTalk/bin/ttdbck* is linked from */usr/sbin/ttdbck. ttdbck* is a database check and recovery tool for the ToolTalk databases.

Table 1-4 Location of ToolTalk Service Files

Directory and File(s)	Description
/usr/ToolTalk/bin/ *tt_type_comp*	*/usr/ToolTalk/bin/tt_type_comp* is linked from */usr/sbin/tt_type_comp*. *tt_type_comp* is a compiler for process types and object types.
/usr/ToolTalk/lib/ *libtt.so* *libtt.a* */usr/ToolTalk/include* *tt_c.h*	The */usr/ToolTalk/libtt** files are linked from the */usr/libtt** symbolic links. */usr/ToolTalk/include/tt_c.h* is linked from */usr/include/tt_c.h.* 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/ToolTalk/man1/ *tt_type_comp1* *ttcopy.1* *ttmv.1* *ttrm.1* *ttrmdir.1* *ttsesion.1* *tttar.1* */usr/ToolTalk/man3/* *tapi.3* */usr/ToolTalk/man8/* *rpc.ttdbserverd.8* *tapi.8* *ttdbck.8* *ttbserved.8*	*usr/ToolTalk/man1/* is linked from */usr/man/u_man/manl/ToolTalk/*. */usr/ToolTalk/man3/* is linked from */usr/man/p_man/man3/ToolTalk/*. */usr/ToolTalk/man8/* is linked from */usr/man/p_man/man8/ToolTalk*. These are the man pages for the ToolTalk binary files, type compiler, enhanced shell commands, API, and database check utility.

Table 1-4 Location of ToolTalk Service Files

Sample Programs

A sample program called *ttsample1_sgi* is provided in the */usr/ToolTalk/examples* directory, along with the makefile *Makefile.sgi* for building it. This program is used in a number of examples throughout this manual. It is similar to the *ttsample1* program delivered in SunSoft™ ToolTalk, but has been modified to work in the X/Motif environment on SGI hosts. *ttsample2_sgi_client* and *ttsample2_sgi_service* combine to form the second example that runs on SGI hosts.

The following sample programs are also referred to in this manual. Although their code is provided, they will not run in the SGI environment and are for illustrative purposes only.

Sun_EditDemo
: This object-oriented XView program creates objects from lines of code. The program includes an object control window and simple editor. Objects are wrapped in text with C-style comments.

ttmon
: This program is a simple message monitioring application.

Chapter 2

Participating in ToolTalk Sessions

This chapter provides instructions on how to participate in a ToolTalk session. It also shows you how to manage 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 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 and started a session with the ToolTalk service, you can join files and user sessions to provide additional information to the ToolTalk service. When your process is ready to quit, you unregister your message patterns and close your ToolTalk session.

Including the ToolTalk API Header File

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

The following code sample shows how the `ttsample1_sgi` program includes this file.

```
/*
 * ttsample1_sgi.c -- dynamic pattern, procedural
 *                    notification  This has been modified
 *                    form the SunSoft version to work
 *                    in the X/Motif environment on SGI
 */

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

```
#include <Xm/Xm.h>
#include <Xm/Form.h>
#include <Xm/PushB.h>
#include <Xm/Scale.h>
#include <Xm/RowColumn.h>

#include <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 2-1.

Return Type	ToolTalk Function
char *	tt_open(void)
int	tt_fd(void)
char *	tt_X_session(const char *xdisplay)
Tt_status	tt_default_session_set(const char *sessid)

Table 2-1 Functions for Registering with ToolTalk Service

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*) and a matching file descriptor (*fd*).

The following code sample initializes and registers the sample program with the ToolTalk service.

```
int ttfd;
```

```
/*
 * Initialize ToolTalk, using the initial default session,
and
 * obtain the file descriptor that will become active
whenever
 * ToolTalk has a message for this process.
 */
my_procid = tt_open();
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.

Note: Your application must call `tt_open()` before other `tt_` calls are made; otherwise, core dumps may occur. However, there are two exceptions: `tt_default_procid_set()` and `tt_X_session()` can be called before `tt_open()`.

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();
```

The following calls are required and must be made in the specified order.

1. `tt_X_session();`

   This call retrieves the name of the session associated with an X11 display server. `tt_X_session()` takes the argument

   `char *xdisplay_name`

   where `xdisplay_name` is the name of an X11 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.

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 that uses *Xt* can have a callback function invoked when the file descriptor becomes active.

```
/*
* Arrange for Xt to call receive_tt_message when the
* ToolTalk file descriptor becomes active.
*/
```

```
Xt_input = XtAppAddInput(app, ttfd,
 (Xtpointer)XtInputReadMask,
 (XtInputCallbackProc)receive_tt_message, NULL)
```

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

Window Toolkits	Code Used
XView	notify_set_input_func()
X Window System Xt (Intrinsics)	XtAddInput(), XtAppAddInput()
TNT	wire_AddFileHandler()
Other 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.

Table 2-2 Code Used to Watch for Arriving Messages

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.

Dynamic Message Patterns

This chapter describes how to provide dynamic 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 listed in the message pattern.

You can provide message pattern information to the ToolTalk service 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 to receive a defined set of messages, the static method provides an easy way to specify the message pattern information. For more information, see Chapter 4, “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 refer to 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 and, to some extent, the number of messages you receive. You can supply multiple values for each attribute you add to a pattern. However, some attributes are set and have only one value.

Table 3-1 provides a complete list of attributes you can put in your message patterns.

Pattern Attributes	Value	Description
Category	TT_OBSERVE, TT_HANDLE,	Declares whether you want to perform the operation listed in a message or only view a message.
Scope	TT_SESSION, TT_FILE, TT_BOTH, TT_FILE_IN_SESSION	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 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.
Object	char *objid	Declares what objects in which you are interested.
Operation	char *opname	Declares what operations in which you are interested.
Otype	char *otype	Declares the type of objects in which you are interested.

Table 3-1 ToolTalk Message Pattern Attributes

address	TT_PROCEDURE, TT_OBJECT, TT_HANDLER, TT_OTYPE	Declares what type of address in which you are interested.
address	TT_PROCEDURE, TT_OBJECT, TT_HANDLER, TT_OTYPE	Declares what type of address in which you are interested.
disposition	TT_DISCARD, TT_QUEUE, TT_START	Instructs the ToolTalk service how to handle undeliverable messages to your application .
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.

Table 3-1 ToolTalk Message Pattern Attributes

All your message patterns must minimally specify:

- 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.

  Note: Messages that have a TT_BOTH scope will match your pattern if it has either TT_FILE or TT_SESSION.
- Category — Whether the application wants to perform operations listed in messages or only view messages.
- Use TT_OBSERVE if you only want to view messages.
- Use TT_HANDLE to if you want to perform operations listed in the message.

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

- If no pattern attribute is specified, the ToolTalk service counts the message attribute as matched. 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.

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 3-2.

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)
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_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_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)
Tt_status	tt_pattern_opnum_add(Tt_pattern p, int opnum)
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)

Table 3-2 ToolTalk Dynamic Message Pattern Attributes

Tt_status	tt_pattern_session_add(Tt_pattern p, const char *sessid)
Tt_status	tt_pattern_state_add(Tt_pattern p, Tt_state s)
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_pattern_destroy(Tt_pattern p)

Table 3-2 ToolTalk Dynamic Message Pattern Attributes

Creating a Message Pattern

To create message patterns, use the `tt_pattern_create()` function. You can use this funtion 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 asumes that you want any value to match. See Table 3-1 for a complete list of pattern attributes.

Note: 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.

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`; 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 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, you 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

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 or file, 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()` join the default sessionThe following code sample shows how to join the default session.

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

tt_session_join(tt_default_session());
```

Table 3-3 lists the ToolTalk functions you use to join the session in which you are interested.

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

Table 3-3 ToolTalk Functions for Joining Default Sessions

Tt_status	tt_session_join(const char *sessid)
Tt_status	tt_session_quit(const char *sessid)

Table 3-3 ToolTalk Functions for Joining Default Sessions

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 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 interested. Table 3-4 lists the ToolTalk functions you use to express your interest in specific files.

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

Table 3-4 ToolTalk Functions for Joining Files of Interest

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

This chapter describes how to provide static 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 listed in 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. See Chapter 3, "Dynamic Message Patterns," for more information.
- If you want to receive a defined set of messages, the static method provides an easy way to specify the message pattern information.

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 refer to 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 and, to some extent, the number of messages you receive. You can supply multiple values for each attribute you add to a pattern. However, some attributes are set and have only one value.

Table 4-1 provides a complete list of attributes you can put in your message patterns.

Pattern Attribute	Value	Description
Category	TT_OBSERVE, TT_HANDLE,	Declares whether you want to perform the operation listed in a message or only view a message.
Scope	TT_SESSION, TT_FILE, TT_BOTH, TT_FILE_IN_SESSION	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 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.
Object	char *objid	Declares what objects in which you are interested.
Operation	char *opname	Declares what operations in which you are interested.
Otype	char *otype	Declares the type of objects in which you are interested.
address	TT_PROCEDURE, TT_OBJECT, TT_HANDLER, TT_OTYPE	Declares what type of address in which you are interested.

Table 4-1 ToolTalk Static Message Pattern Attributes

Pattern Attribute	Value	Description
disposition	TT_DISCARD, TT_QUEUE, TT_START	Instructs the ToolTalk service how to handle undeliverable messages to your application.
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.

Table 4-1 ToolTalk Static Message Pattern Attributes

All your message patterns must minimally specify:

- 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.

  Note: Messages that have a TT_BOTH scope will match your pattern if it has either TT_FILE or TT_SESSION.
- Category — Whether the application wants to perform operations listed in messages or only view messages.
  - Use `TT_OBSERVE` if you only want to view messages.

- Use `TT_HANDLE` to if you want to perform operations listed in the message.

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

- If no pattern attribute is specified, the ToolTalk service counts the message attribute as matched. 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.

This chapter describes how to provide process and object type information to Static Message Patterns.

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 type information, the ToolTalk service creates message patterns based on the information. 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 file with the ToolTalk service, the message patterns listed in it are automatically registered, too.

The ptype provides application information that the ToolTalk service can use when the application is not running. This information is used to start your processes if necessary, to receive a message, queue messages until the process starts, or deliver `TT_PROCEDURE`-addressed messages to your process.

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 procedure and process messages that the program wants to receive. Messages to be observed are described separately from messages to be handled.

Signatures

Signatures describe the procedure and process 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 makes incoming messages easy to identify.

Creating a Ptype File

The following code example shows the syntax for a ptype file

```
ptype::=’ptype’ ptid ‘{’
      property*
      [‘observe:’ psignature*]
      [‘handle:’ psignature* ]
      ‘}’ [‘;’]
property::=property_id value ‘;’
property_id::=‘per_file’
   |‘per_session’
   |‘start’
value::=string
   |number
ptid::= identifier
psignature::=[scope] op args
       [‘=>’
       [‘start’][‘queue’]
       [‘opnum=’number]]
       ‘;’
scope:=‘file’
   |‘session’
   |‘file_in_session’
args::=‘(‘ argspec {, argspec}* ‘)’
   |‘(void)’
   |‘()’
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, such as `Sun_EditDemo`. The ptid cannot exceed 32 characters and should not be one of the reserved identifiers: `ptype`, `otype`, `per_file`, `per_session`, `start`, `opnum`, `queue`, `file`, `session`, `observe`, or `handle`.

`per_file`

The maximum number of processes of this type that can concurrently observe a particular file. If the maximum number processes of this type are already observing a file, the ToolTalk service will not start another one.

Set this limit to 1 for tools that cannot handle multiple processes updating the same file.

`per_session`

The maximum number of processes of this type that can concurrently run in a single session. If maximum number of processes of this type are already running in a session, the ToolTalk service will not start another one.

Set this limit to 1 for tools that manage multiple files in one process. All files will then be handled by a single process in each user session.

`start`

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

Before executing the command, the ToolTalk service defines `_SUN_TT_FILE` as an environment variable with the value of the file attribute of the message that caused the application to be started. The `_SUN_TT_FILE` value has the following format: *hostname:pathname*, even though the file attribute is a pathname. 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.

Psignature Matching Information

`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 matches without regard to the arguments.

`scope`

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

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. The opnum enables you to specify an operation more than once and list unique arguments with each instance of the operation.

The following sample code illustrates a ptype file.

```
#include "Sun_EditDemo_opnums.h"

ptype Sun_EditDemo {
   /* setenv SUN_EDITDEMO_HOME to install dir for the demo */
    start"${SUN_EDITDEMO_HOME}/edit";
    handle:
    /* edit file named in message, start editor if necessary
*/
```

```
    session Sun_EditDemo_edit(void)
                => start opnum=SUN_EDITDEMO_EDIT;

    /* tell editor viewing file in message to save file */
    session Sun_EditDemo_save(void)
                => opnum=SUN_EDITDEMO_SAVE;

    /* save file named in message to new filename */
    session Sun_EditDemo_save_as(in string new_filename)
                => opnum=SUN_EDITDEMO_SAVE_AS;
    /* bring down editor viewing file in message */
    session Sun_EditDemo_close(void)
                => opnum=SUN_EDITDEMO_CLOSE;
};
```

The `Sun_EditDemo_opnums.h` file defines symbolic definitions for all the opnums used by `edit.c`, allowing both the `edit.types` file and `edit.c` file to share the same definitions.

The following sample code illustrates a `ptype` file for the `ttsample2_sgi` example:

```
/* ttsample2_sgi_ptypes - ptype definitions for
ttsample2_sgi example */
#include “ttsample2_sgi_opnums.h”
ptype TTSAMPLE2
{
 start “/usr/tmp/ttsample2_sgi_service”;
 handle:
    session ttsample2_value(in int send, out int receive)
                      => start opnum=TTSAMPLE2_VALUE;
};
```

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*) file. An otype file contains the ptype information of the application that manages the object and 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 which 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 code sample shows the syntax for an otype file.

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

osignature     ::= op args [rhs][inherit] ';'
rhs            ::= ['=>' ptid [scope]]
                   ['start']['queue']
                   ['opnum='number]
inherit        ::= 'from' otid
args           ::= '(' argspec {, argspec}* ')'
               |   '(void)'
               |   '()'
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, per_file, per_session, 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.

`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. The opnum enables you to specify an operation more than once and list unique arguments with each instance of the operation.

`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.

The following sample code illustrates an otype file.

```
Include "Sun_EditDemo_opnums.h"
otype Sun_EditDemo_object {
   handle:
   /* hilite object given by objid, starts editor
      if necessary */
   hilite_obj(in string objid)
   => Sun_EditDemo session start
opnum=SUN_EDITDEMO_HILITE_OBJ;
};
```

The `Sun_EditDemo_opnums.h` file defines symbolic definitions for all the opnums used by `edit.c`, allowing both the `edit.types` file and `edit.c` file to share the same definitions.

Installing Type Information

The 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 types database.

- To receive messages addressed to objects your application creates and manages, the otype definitions must also be installed in the types database.

To place your type information into the xdr format 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 the types database definitions for your type information and stores them in the types database.

To install an application's ptype and otype, 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 the types database.

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>
```

For more information on `tt_type_comp`, see the man page.

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 XDR 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`.

Making Type Information Available to the ToolTalk Service

After you run `tt_type_comp`, you need to make your type information available to the ToolTalk service. To do this, tell the ToolTalk service to read the type information in the types database as follows:

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

   `% ps -elf | grep ttsession`

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

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

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 register your ptype information with the ToolTalk service, use `tt_ptype_declare()` during your application's ToolTalk initialization routine. The ToolTalk service will read the type information and 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 information 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()`.

The following code sample registers its ptype during its `edit.c` program 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("Sun_EditDemo") != TT_OK) {
                fprintf
          (stderr,"Sun_EditDemo is not an installed
              ptype.\n");
                return 0;
        }
        ttfd = tt_fd();
        notify_set_input_func(edit_ui_base_window,
                      (Notify_func)edit_receive_tt_message,
                       ttfd);

        tt_session_join(tt_default_session());

        /*
         * 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;
}
```

The following code sample registers its ptype during its `ttsample2_sgi_service.c` program initialization.

```
/* declare ptype
 * note there is no dynamic pattern registration
 */
if (tt_ptype_declare("TTSAMPLE2") != TT_OK)
   {
    /* TTSAMPLE2 is not an installed ptype */
    fprintf(stderr, "No TTSAMPLE2 type for ToolTalk\n");
    exit(-1);
   }
```

Data Type Registration

Your application will use either process-oriented messages or object-oriented messages to communicate with other applications. To send messages to types of processes, types of objects, and specific objects, ptypes (for process-oriented messages) and otypes (for object-oriented messages) are required. This type information is compiled at application installation time and stored in the types database.

To communicate with other vendors' applications, you must know their ptypes and possibly their otypes (depending on the method of messaging you choose).

Chapter 5

Sending Messages

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 5-1.

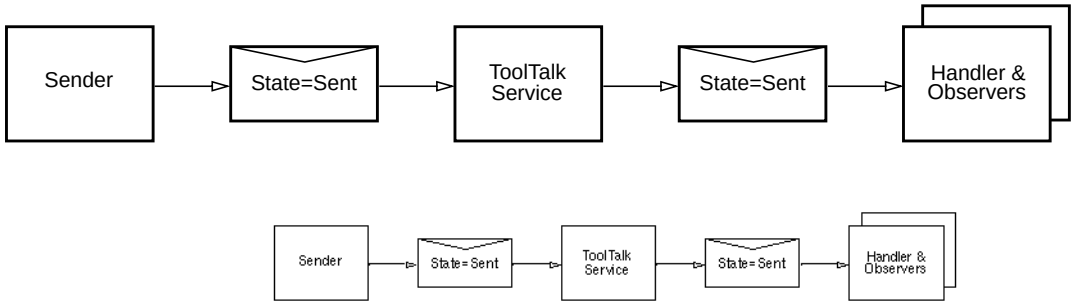


Figure 5-1 Notice Routing

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 side trip to observers. Figure 5-2 illustrates the request routing procedure.

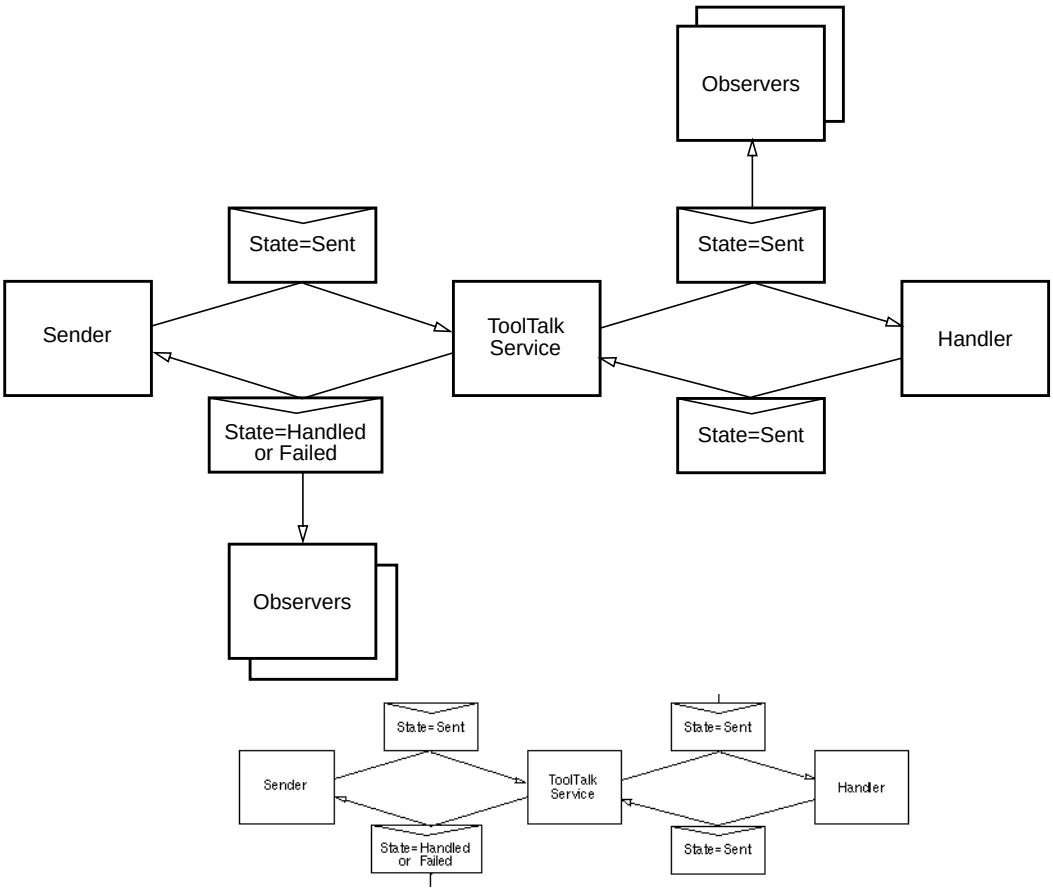


Figure 5-2 Request Routing

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 both; observers absorb a request without sending it back.

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 5-1.

Message Attribute	Value	Description	Who Can Complete
Arguments	arguments or results	Arguments used in the operation. If the message is a reply, this field contains the results of the operation.	Sender, replier
Class	TT_NOTICE, TT_REQUEST	Specifies whether the recipient needs to perform an operation.	Sender
File	char *pathname	The file involved in the operation.	Sender, ToolTalk
Object	char *objid	The object involved in the operation.	Sender, ToolTalk
Operation	char *opname	Name of operation to be performed.	Sender
Otype	char *otype	The type of object involved in the operation.	Sender, ToolTalk
Address	TT_PROCEDURE, TT_OBJECT, TT_HANDLER, TT_OTYPE	Where the message should be sent.	Sender
Handler	char *procid	The receiving process.	Sender, ToolTalk
Handler_ptype	char *ptype	The type of receiving process.	Sender, ToolTalk

Table 5-1 ToolTalk Message Attributes

Message Attribute	Value	Description	Who Can Complete
Disposition	`TT_DISCARD, TT_QUEUE, TT_START`	Specifies what ti de if the message cannot be received by any running process.	Sender, ToolTalk
Scope	TT_SESSION, TT_FILE, TT_BOTH, TT_FILE_IN_SESSION	Applications that will be considered as potential recipients based on their registered interest in a session or file.	Sender, ToolTalk
Sender_ptype	`char * ptype`	The type od the sending process.	Sender, ToolTalk
Session	char *sessid	The sending process's session.	Sender, ToolTalk
Status	int status, char *status_str	Additional information about the state of the message.	Replier, ToolTalk

Table 5-1 ToolTalk Message Attributes

Address Attribute

Messages addressed to other applications can be addressed to a particular process or 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.

Session Scope

When a message is scoped to a session, only those applications that have joined the session are considered as potential recipients.

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.

ToolTalk Message Delivery Algorithm

To help you further 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 many 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.

   If the address is `TT_PROCEDURE`, 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.

To speed up dispatch, the sender may fill in the *handler_ptype* if known. However, this 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 message patterns 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 registed 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/gondor/joe/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 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 spreadsheet application, 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. Agnes 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:
   - 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`*.
   - 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`.
   - 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 as it is looking at the wrong file.
   - 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.

Note: The code samples in this chapter are fragments from the sample programs, ttsample1_sgi and Sun_EditDemo. The following source files reside in the /usr/ToolTalk/examples/ directory.

```
ttsample1_sgi.c
SunEdit_Demo_opnums.h
edit.types.model
edit.c
cntl.c
```

Creating Messages

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

- General-purpose function
  - tt_message_create()
- Process-oriented notice and request functions
  - tt_pnotice_create()
  - tt_prequest_create()
- 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 5-2 is a list of theToolTalk functions that are used to create and complete messages.

Return Type	ToolTalk Function
Tt_message	tt_onotice_create(const char *objid, const char *op)
`Tt_message`	`ttt_orequwst_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_status	tt_message_address_set(Tt_message m, Tt_address p)
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, const 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_barg_add(Tt_message m, Tt_mode n, const char *vtype, const unsigned char *value, int len)
Tt_status	tt_message_iarg_add(Tt_message m, Tt_mode n, const char *vtype, int value)
Tt_status	tt_message_class_set(Tt_message m, Tt_class c)

Table 5-2 Functions Used to Create and Complete Messages

Return Type	ToolTalk Function
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_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_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)

Table 5-2 Functions Used to Create and Complete Messages

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. 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. Usually, one process makes a general request, picks the handler attribute from the reply, and directs further messages to that handler. This enables two processes to rendezvous through broadcast message passing and then go into a dialogue.

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`, `TT_OUT`, 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`. `TT_OUT` indicates that the argument is written by the handler and read by the sender.

`vtype`
: 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:

- 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).

- 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:

- Objid

  Fill in the unique object identifier.

- 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.

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

```
Tt_callback action
sample_msg_callback(Tt_message m, Tt_pattern p)
{
   ... process the 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:
         xv_set(cntl_ui_base_window, FRAME_LEFT_FOOTER,
            "Starting editor...", NULL);
```

```
      break;
   case TT_HANDLED:
      xv_set(cntl_ui_base_window, FRAME_LEFT_FOOTER,
"",NULL);
      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)));
    }
    xv_set(cntl_ui_base_window,
FRAME_LEFT_FOOTER,msg,NULL);
          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;
}
```

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 that you will want to compare against your request, do not destroy the message until you have handled the reply.

Request Examples

The following code sample from *ttsample1_sgi* 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,
    "int", NULL);

  XtVaGetValues(slider, XmNvalue, &slider_val, NULL);

 tt_message_arg_ival_set(msg_out, 0, slider_val);
 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);
```

The following code sample illustrates 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) {
      xv_set(cntl_ui_base_window, FRAME_LEFT_FOOTER,
       “No object id selected”, NULL);
      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);
}
```

Chapter 6

Receiving Messages

This chapter describes how to retrieve messages 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 be able to invoke 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.

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: 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.

The following code sample from *ttsample1_sgi* illustrates how to receive a message.

```
/*
 * When a ToolTalk message is available, receive it; if it's
 * a ttsample1_value message, update the gauge with the new
 * value.
 */
```

```
void receive_tt_message()
{
   Tt_message msg_in;
   int mark;
   int val_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;
```

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 3, "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 sent by you:

- Place specific callbacks on requests before you send them with `tt_message_callback_add()`. See Chapter 5, "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.
- Place information meaningful to your application on the request with the `tt_message_user_set()` 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 sample code from *ttsample1_sgi* allocates storage, examines a message's contents, and releases the storage.

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

mark = tt_mark();

op = tt_message_op(msg_in);
err = tt_ptr_error(op);
if (err > TT_WRN_LAST) {
   printf( "tt_message_op(): %s\n", tt_status_message(err));
} else if (op != 0) {
   if (0==strcmp("ttsample1_value", tt_message_op(msg_in))) {
      tt_message_arg_ival(msg_in, 0, &val_in);
      XtVaSetValues(gauge, XmNvalue, val_in, NULL);
   }
}

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

Table 6-1 lists the ToolTalk functions you use to examine the attributes of a message you have received.

Return Type	ToolTalk Function
Tt_address	tt_message_address(Tt_message m)
`Tt_staus`	`tt_message_arg_bval(Tt_message m, intn, usigned char **value, int *len)`

Table 6-1 Function to Examine Message Attributes

Return Type	ToolTalk Function
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)
char *	tt_message_arg_val(Tt_message m, int n)
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_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)

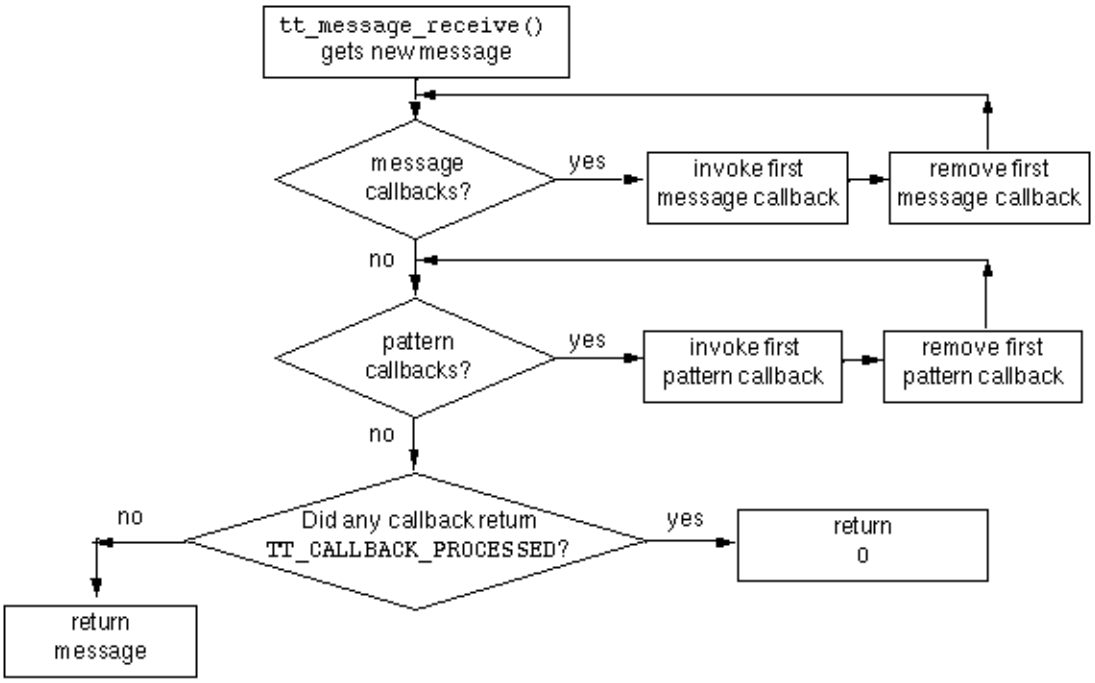
Table 6-1 Function to Examine Message Attributes

Return Type	ToolTalk Function
uid_t	tt_message_uid(Tt_message m)
void *	tt_message_user(Tt_message m, int key)

Table 6-1 Function to Examine Message Attributes

Invoking Callback Routines

Figure 6-1 illustrates how the ToolTalk service invokes message and pattern callbacks when `tt_message_receive()` is called to retrieve a new message.



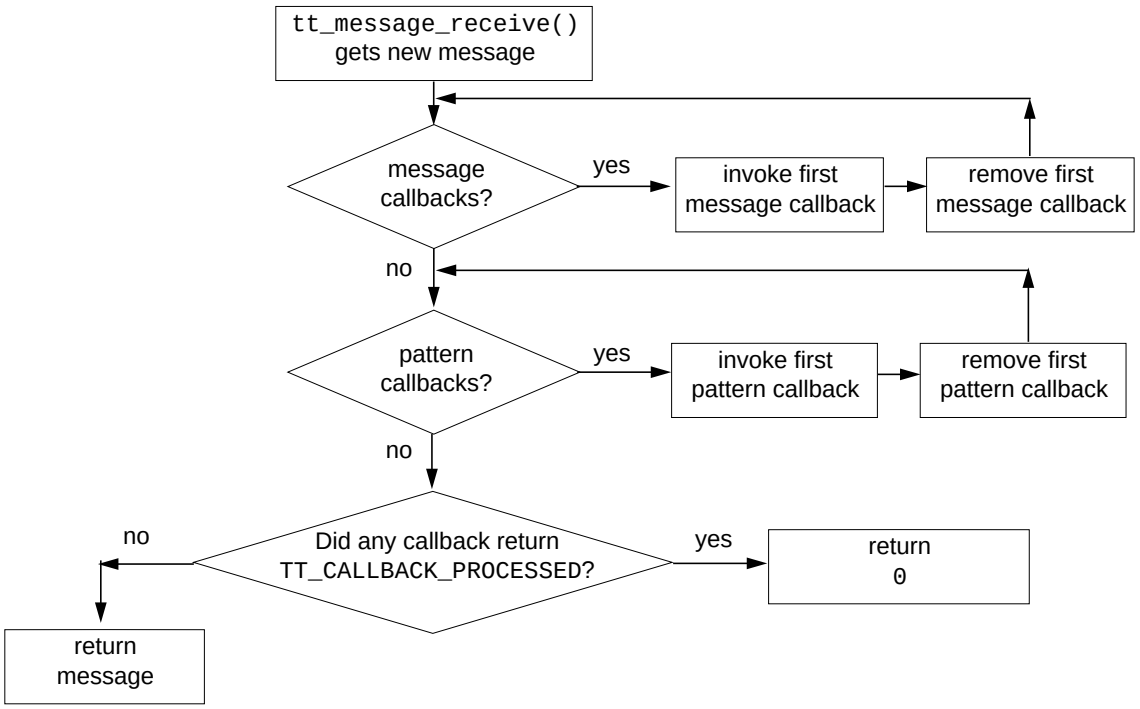


Figure 6-1 How Callbacks Are Invoked

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 6-2 lists the ToolTalk functions you use to reply to requests.

Return Type	ToolTalk Function
Tt_mode	tt_message_arg_mode(Tt_message m, int n)
`Tt_status`	`tt_message_agr_bval_set(Tt_message m, intn, const 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_reply(Tt_message m)

Table 6-2 Functions to Reply to Requests

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 6-3 to reject or fail a request.

Return Type	ToolTalk Function
Tt_status	tt_message_reject(Tt_message m)
Tt_status	tt_message_fail(Tt_message m)
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)

Table 6-3 Rejecting or Failing Requests

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.

Chapter 7

Objects

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 types database. See Chapter 4, “Static Message Patterns,” for information on otypes.

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

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 7-1.

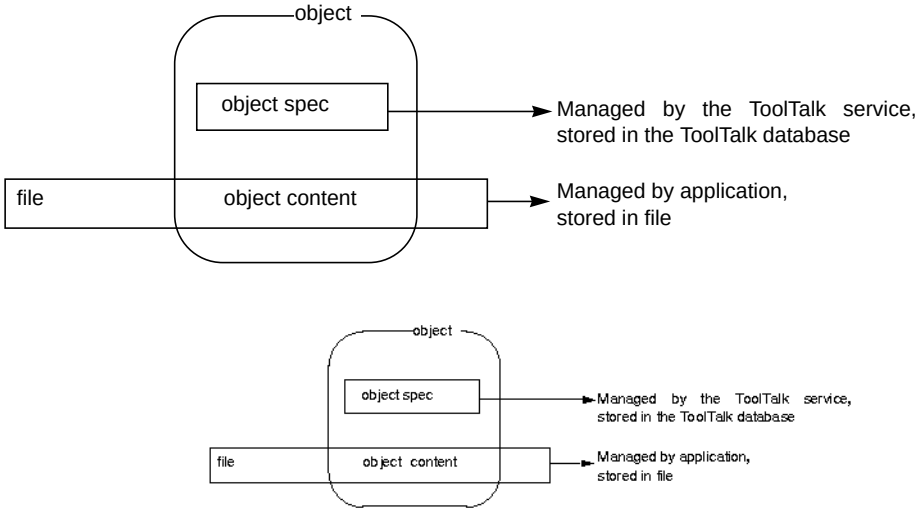


Figure 7-1 ToolTalk Object Data

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: a paragraph, a source code function, or a range of spreadsheet cells, for example.

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 of 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 7-1 lists the ToolTalk functions you use to create and write object spec.

Return Type	ToolTalk Function
`char *`	`tt_spec_create(const char *filepath)`
`Tt_status`	`tt_spec_prop_set(const char *objid, const char *propname, const char *value)`
`Tt_status`	`tt_spec_prop_add(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)`

Table 7-1 Functions to Create

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 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 types database.)

Determining Object Specification Properties

You can determine what *properties* you want associated with an object; you add 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. A user 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 the user.

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 7-2 lists the ToolTalk functions you use to maintain object specs.

Return Type	ToolTalk Function
char *	tt_spec_file(const char *objid)
char *	tt_spec_type(const char *objid)
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_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_objid_objkey(const char *objid)

Table 7-2 Functions to Maintain Object Specifications

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_spec_move(const char *objid, const char *newfilepath)

Table 7-2 Functions to Maintain Object Specifications

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 sample code 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--) {
   xv_set(cntl_ui_olist, PANEL_LIST_DELETE, i, NULL);
}


list_item = 0;
file = (char *)xv_get(cntl_ui_file_field, PANEL_VALUE);
if (tt_file_objects_query(file,
        if (tt_file_objects_query(file,

(Tt_filter_function)cntl_gather_specs,
                                  &list_item, NULL) != TT_OK)

{
     xv_set(cntl_ui_base_window, FRAME_LEFT_FOOTER,
            “Couldn’t query objects for file”, NULL);
     return 0;
   }

   return 1;
}
```

Within the `tt_file_objects_query()` function, the application calls `cntl_gather_specs`, a filter function that inserts objects into a scrolling list.

```
/*
 * Function to insert the objid given into the scrolling
lists
```

```
 * of objects for a file. Used inside tt_file_objects_query
as
 * it iterates through all the ToolTalk objects in a file.
 */
Tt_filter_action
cntl_gather_specs(objid, list_count, acc)
     char *objid;
     void *list_count;
     void *acc;
{
   int *i = (int *)list_count;

   xv_set(cntl_ui_olist, PANEL_LIST_INSERT, *i,
       PANEL_LIST_STRING, *i, objid,
       NULL);

   *i = (*i + 1);

    /* continue processing */
    return TT_FILTER_CONTINUE;
}
```

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 by removing, moving, or renaming files with UNIX commands such as rm or mv. Broken references will result in undeliverable messages.

Managing Files that Contain Object Data

To keep the ToolTalk database that services the disk partition where a file that contains object data is stored up-to-date, use the ToolTalk functions to copy, move, or destroy the file. Table 7-3 lists the ToolTalk functions you use to manage files that contain object data.

Return Type	ToolTalk Functions
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)

Table 7-3 Functions to Copy, Move, or Remove Files that Contain Object Data

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 7-4 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.

Command	Description
ttcopy	Copies file to new location. Updates file and object location information in ToolTalk database.
ttmv	Moves directory or files to new location. Updates file and object location information in ToolTalk database. Removes old version of file or directory.
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.

Table 7-4 ToolTalk-Wrapped Shell Commands

Examples

The following 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 who observe messages with the `Sun_EditDemo_new_object` operation. If other applications are displaying a list of objects in a file managed by `Sun_EditDemo`, they update their list after receiving this notice.

```
/*
 * Make a ToolTalk spec out of the selected text in this
 * textpane. Once the spec is successfully created and
written
 * to a database, wrap the text with C-style comments in
order
 * to delimit the object and send out a notification that an
 * object has been created in this file.
 */
Menu_item
edit_ui_make_object(item, event)
   Panel_itemitem;
   Event*event;
{
         int               mark = tt_mark();
   char*objid;
   char*file;
   char*sel;
   Textsw_indexfirst, last;
   charobj_start_text[100];
   charobj_end_text[100];
   Tt_message msg;

   if (! get_selection(edit_ui_xserver, edit_ui_textpane,
          &sel, &first, &last)) {
      xv_set(edit_ui_base_window, FRAME_LEFT_FOOTER,
             “First select some text”, NULL);
      tt_release(mark);
      return item;
   }
   file = tt_default_file();

   if (file == (char *)0) {
      xv_set(edit_ui_base_window, FRAME_LEFT_FOOTER,
             “Not editing any file”, NULL);
      tt_release(mark);
      return item;
   }

   /* create a new spec */

   objid = tt_spec_create(tt_default_file());
   if (tt_pointer_error(objid) != TT_OK) {
      xv_set(edit_ui_base_window, FRAME_LEFT_FOOTER,
             “Couldn’t create object”, NULL);
```

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

   /* set its otype */

   tt_spec_type_set(objid, “Sun_EditDemo_object”);
   if (tt_spec_write(objid) != TT_OK) {
      xv_set(edit_ui_base_window, FRAME_LEFT_FOOTER,
             “Couldn’t write out object”, NULL);
      tt_release(mark);
      return item;
   }

   /* wrap spec’s contents (the selected text) with C-style
*/
   /* comments. */

   sprintf(obj_start_text,” /* begin_object(%s) */”, objid);
   sprintf(obj_end_text,”   /* end_object(%s) */”, objid);
   (void)wrap_selection(edit_ui_xserver, edit_ui_textpane,
            obj_start_text, obj_end_text);

   /* now send out a notification that we’ve added a new
object */

   msg = tt_pnotice_create (TT_FILE_IN_SESSION,
                            ”Sun_EditDemo_new_object”);
   tt_message_file_set(msg, file);
   tt_message_send(msg);

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

Chapter 8

Managing Information Storage

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 8-1 to manage the storage of information in the ToolTalk API allocation stack:

Return Type	Tool Talk Functions
int	tt_mark(void)
void	tt_release(int mark)
caddr_t	tt_malloc(size_t s)
void	tt_free(caddr_t p)

Table 8-1 Managing TiilTalk Storage

`tt_mark()` marks information returned by a series of functions.

`tt_release()` frees information returned by a series of functions.

`tt_malloc()` reserves a specified amount of storage in the allocation stack for your use.

`tt_free()` frees storage set aside by `tt_malloc()`.

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 from *ttsample1_sgi* 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();

op = tt_message_op(msg_in);
err = tt_ptr_error(op);
if (err > TT_WRN_LAST) {
   printf( "tt_message_op(): %s\n", tt_status_message(err));
} else if (op != 0) {
   if (0==strcmp("ttsample1_value", tt_message_op(msg_in))) {
      tt_message_arg_ival(msg_in, 0, &val_in);
      XtVaSetValues(gauge, XmNvalue, val_in, NULL);
   }
}

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 alloted 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 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 you 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.

Chapter 9

Handling Errors

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*

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 9-1 to retrieve error values.

Return Type	Tool Talk Functions
char *	tt_status_message(Tt_status ttrc)
Tt_status	tt_int_error(int return_val)

Table 9-1 Retrieving ToolTalk Error Status

Checking ToolTalk Error Status

You can use the ToolTalk error macros shown in Table 9-2 to check error values

Return Type	ToolTalk Macros
Tt_status	tt_ptr_error(pointer)
Tt_status	tt_is_err(pointer)

Table 9-2 ToolTalk Error Macros

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 in the following code example.

```
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 contains 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 sample code 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.
    */
   update_my_spec_ids(old_spec_id, new_spec_id);
   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 values. The `tt_int_error()` function returns a status of `TT_OK` if the value is not very out-of-bounds.

If a value is very 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 following sample code 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));
}
```

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)..
```

Chapter 10

ToolTalk API

This chapter provides reference information for these components of the ToolTalk API:

- “ToolTalk Enumerated Types”
- “ToolTalk Functions”

The enumerated types and functions are listed in alphabetical order in each section.

ToolTalk Enumerated Types

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”

Tt_address

`Tt_address` indicates which message attributes form the address to which the message will be delivered. Possible values and meanings are:

`TT_HANDLER`
: Addressed to a specific handler that ca 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. Possible values and their meanings are:

`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. Possible values and meanings are:

`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. Possible values and meanings are:

`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. Possible values and their meanings are:

`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. Possible values and meanings are:

`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. Possible values are:

`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. Possible values and meanings are:

`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. Possible values and their meanings are:

`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 2, "Participating in ToolTalk Sessions," for instructions on determining whether the `Tt_status` code is a warning or an error and for retrieving the catalog string for a `Tt_status` code.

Appendix D, "ToolTalk Error Messages," lists the `Tt_status` codes. The following information is provided for each status code:

- Message id
- Catalog string
- Meaning
- Remedy

ToolTalk Functions

tt_close

```
Tt_status    tt_close(void)
```

Closes the current default process identifier (procid).

Note: `tt_close()` should be the last ToolTalk function the process calls.

Returned Value

`Tt_status`

The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_PROCID
```

Related Functions

`tt_open()`

tt_default_file

Returns the current default file. Joining a file makes it the default file.

Returned Value

char *

Pointer to a character string specifying the current default file. If the pointer is `NULL`, no default is set. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_PROCID
```

Related Functions

`tt_file_join()`

tt_default_file_set

```
Tt_status    tt_default_file_set(const char *docid)
```

Sets the default file to the specified file.

Arguments

`const char *docid`
: Pointer to a character string specifying the file you want as the default file.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_PROCID
TT_ERR_FILE
```

tt_default_procid

```
char         *tt_default_procid(void)
```

Retrieves the current default process identifier (procid) for your process. The procid is used in the sender field of messages.

Returned Value

`char *`
: Pointer to character string that uniquely identifies the current default process. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid.

  Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_PROCID
```

tt_default_procid_set

```
Tt_status    tt_default_procid_set(const char *procid)
```

Sets the current default procid. The default procid is set by `tt_open()`. Only processes that do multiple `tt_open()` calls and juggle multiple procids ever need to use this function.

Arguments

`const char *procid`
: Name of process you want to set up as the default process.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

  ```
  TT_OK
  TT_ERR_NOMP
  TT_ERR_PROCID
  ```

Related Functions

`tt_open()`

tt_default_ptype

```
char          *tt_default_ptype(void)
```

Retrieves the current default process type (ptype). Declaring a ptype makes it the default ptype. The default ptype is used in the sender ptype field of your message.

Returned Value

`char *`
: Pointer to character string that uniquely identifies the current default process type. If the pointer is `NULL`, no default is set. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

  ```
  TT_OK
  TT_ERR_NOMP
  TT_ERR_PROCID
  ```

Related Functions

`tt_ptype_declare()`

tt_default_ptype_set

```
Tt_status    tt_default_ptype_set(const char *ptid)
```

Sets the default process type (ptype) to the provided string.

Arguments

`const char *ptid`
: Use the character string that uniquely identifies the process type you wish to set up as the default process type.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

  ```
  TT_OK
  TT_ERR_NOMP
  TT_ERR_PROCID
  ```

tt_default_session

```
char         *tt_default_session(void)
```

Retrieves the current default session identifier from the ToolTalk service for the current default procid.

Returned Value

`char *`
: Pointer to the unique identifier for the current session. If the pointer is `NULL`, no default is set. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

  ```
  TT_OK
  TT_ERR_NOMP
  TT_ERR_PROCID
  ```

Related Functions

`tt_default_procid()`

tt_default_session_set

```
Tt_status    tt_default_session_set(const char *sessid)
```

Sets the current default session identifier for the current default procid.

Note: The ToolTalk service uses the initial user session as the default session and supports one session per procid. To join other sessions, your program must first set the new session as the default and then initialize and register. The calls required must be in this order: `tt_default_session_set`, `tt_open`, `tt_fd`

Arguments

`const char *sessid`
: Pointer to the unique identifier for the session in question.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_PROCID
TT_ERR_SESSION
```

Related Functions

`tt_open()`
`tt_fd()`

tt_error_int

```
int          tt_error_int(Tt_status ttrc)
```

Given a `Tt_status` code, returns an integer error object encoding the code.

Note: The integer error objects are negative integers, so only use this when the valid integer values are non-negative.

Arguments

`Tt_status ttrc`
: `Tt_status` code you want to encode.

Returned Value

int
: Encoded `Tt_status` code.

tt_error_pointer

void `*tt_error_pointer(Tt_status ttrc)`

Given a `Tt_status` code, returns a pointer to an error object encoding the code.

Arguments

`Tt_status ttrc`
: `Tt_status` code you want to encode.

Returned Value

`void *`
: Pointer to encoded `Tt_status` code.

tt_fd

`int tt_fd(void)`

Returns a file descriptor (fd) which is used to alert your program that a message has arrived for the default procid in the default session. File descriptors are either active or inactive. When your file descriptor becomes active, you need to call `tt_message_receive`.

Note: You must have a separate file descriptor for each procid. Each time you call `tt_open`, use `tt_fd` to get an associated file descriptor.

Returned Value

`int`

File descriptor for your current `procid`. Use `tt_int_error()`, which returns `Tt_status`, to determine if the integer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_PROCID
TT_ERR_SESSION
```

Related Functions

```
tt_open()
tt_message_receive()
```

tt_file_copy

```
Tt_status    tt_file_copy(const char *oldfilepath, const char
             *newfilepath)
```

Copies all the objects on the specified file to the new file. Any objects already on the second file are not removed.

Arguments

`const char *oldfilepath`

Pointer to the name of the file whose objects are to be copied.

`const char *newfilepath`

Pointer to the name of the file on which to create the copied objects.

Returned Value

`Tt_status`

The status of the operation. Possible values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_FILE
```

```
TT_ERR_NOMP
TT_ERR_PATH
TT_ERR_POINTER
```

Related Functions

```
tt_file_move()
tt_file_destroy()
```

tt_file_destroy

```
Tt_status    tt_file_destroy(const char *filepath)
```

Removes all the objects on the files and directories rooted at filepath from the appropriate ToolTalk database. Call this function when you `unlink` a file or `rmdir` a directory.

Arguments

`const char *filepath`
: Pointer to the pathname of the file to be removed.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_ACCESS
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_FILE
TT_ERR_NOMP
TT_ERR_PATH
TT_ERR_POINTER
```

Related Functions

```
tt_file_copy()
tt_file_move()
rmdir(2)
unlink(2)
```

tt_file_join

```
Tt_status    tt_file_join(const char *filepath)
```

Informs the ToolTalk service that your process is interested in messages involving the file named by the provided string. The ToolTalk service adds this file value to any currently registered patterns with scope `TT_FILE`. The named file becomes the default file.

Arguments

`const char *filepath`
: Pointer to the pathname of the file to be joined.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_PATH
```

tt_file_move

```
Tt_status    tt_file_move(const char *oldfilepath, const char
             *newfilepath)
```

Destroys all the objects on the files and directories rooted at the new filepath, and then moves all the objects on the first file to the second file.

If `oldfilepath` and `newfilepath` are in the same filesystem, then `tt_file_move()` replaces oldfilepath with newfilepath in the path associated with every object in that filesystem. That is, it picks up all the objects in the directory tree rooted at oldfilepath, and overlays them onto newfilepath. In this mode, `tt_file_move()` is like the system call `rename(2)`.

If oldfilepath and newfilepath are on different file systems, neither may be a directory.

Arguments

`const char *oldfilepath`
: The name of the file or directory whose objects are to be moved.

`const char *newfilepath`
: The name of the file or directory to which the objects are to be moved.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_ACCESS
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_FILE
TT_ERR_NOMP
TT_ERR_PATH
TT_ERR_POINTER
```

Related Functions

`tt_file_copy()`
`tt_file_destroy()`
`rename(2)`

tt_file_objects_query

```
Tt_status    tt_file_objects_query(const char *filepath,
             Tt_filter_function filter, void *context, void
             *accumulator)
```

Instructs the ToolTalk service to find all the objects in the named file and pass back the objids to the filter function you created. The context pointer and accumulator pointer you initially specify will also be passed to your filter function.

As the ToolTalk service finds each object, it calls your filter function, passing the objid of the object and the two application-supplied pointers. Your filter

function performs its computation, and returns a `Tt_filter_action` value to tell the query function whether to continue or to stop. `Tt_filter` action values are:

```
TT_FILTER_CONTINUE
TT_FILTER_STOP
```

Arguments

`const char *filepath`
File name.

`Tt_filter_function filter`
Your filter function. `Tt_filter_function` is a typedef "`Tt_filter_action (*) (char *objid, void *context, void *accumulator)`".

`void *context`
A pointer to any information your filter needs to execute. The ToolTalk service does not interpret this argument. It passes it straight through to your filter function.

`void *accumulator`
A pointer to a place for your filter to store the results of the query and filter operations. The ToolTalk service does not interpret this argument. It passes it straight through to your filter function.

Returned Value

`Tt_status`
The status of the operation. Possible values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_PATH
TT_WRN_STOPPED
```

tt_file_quit

```
Tt_status    tt_file_quit(const char *filepath)
```

Informs the ToolTalk service that your process is no longer interested in messages involving the file named by the provided string. The ToolTalk service removes this file value from any currently registered patterns with scope `TT_FILE`. The default file is nulled.

Arguments

`const char *filepath`
File name.

Returned Value

`Tt_status`
The status of the operation. Possible values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_PATH
```

tt_free

```
void         tt_free(caddr_t p)
```

Frees this storage from the ToolTalk API allocation stack.

You may find `tt_free` more convenient than using `tt_mark` and `tt_release` if your application is in a loop obtaining strings from the ToolTalk service and processing each in turn.

Arguments

Related Functions

`tt_malloc()`

tt_initial_session

```
char         *tt_initial_session(void)
```

Returns the session in which the process was created. This is either a process tree session or the X session associated with the display named in the `DISPLAY` environment variable.

Returned Value

`char *`

Identifier for the current ToolTalk session. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

`TT_OK`
`TT_ERR_NOMP`

tt_int_error

```
Tt_status    tt_int_error(int return_val)
```

Given an integer, returns `TT_OK` if the integer is not an error object or the encoded `Tt_status` value if the integer is an error object.

Arguments

`int return_val`

Integer returned by a ToolTalk function.

Returned Value

`Tt_status`

The status of the operation. Possible values are:

`TT_OK`
`TT_`*`xxx`*

tt_is_err

```
int          tt_is_err(Tt_status s)
```

A macro that tells you if the `Tt_status` enum you provided is a warning or an error. `tt_is_err()` expands to `(TT_WRN_LAST < (p))`.

Arguments

`Tt_status s`
: The `Tt_status` code you want to check.

Returned Value

`int`
: If you receive 1, the `Tt_status` enum is an error. If you receive 0, the Tt_status enum is either a warning or `TT_OK`.

tt_malloc

```
caddr_t      tt_malloc(size_t s)
```

Allocates storage on the ToolTalk API allocation stack.

This capability is provided so that your application-provided callback routines can take advantage of the allocation stack. For example, a query filter function might allocate storage to hold a result.

Arguments

`size_t s`
: The amount of storage you want in bytes.

Returned Value

`caddr_t`
: Storage in the ToolTalk API allocation stack given to your application. If `NULL` is returned, no storage is available.

Related Functions

`tt_free()`

tt_mark

```
int          tt_mark(void)
```

Marks a storage position in the ToolTalk API allocation stack. Your application typically does this at the beginning of a procedure.

Returned Value

`int`

Integer that marks your application's storage position in the ToolTalk API allocation stack.

Related Functions

`tt_release()`

tt_message_address

```
Tt_address    tt_message_address(Tt_message m)
```

Retrieves the address attribute from the specified message.

Arguments

`Tt_message m`

Opaque handle for the message involved in this operation.

Returned Value

`Tt_address`

specifies which message attributes form the address of this message. Possible values are:

```
TT_PROCEDURE
TT_OBJECT
TT_HANDLER
TT_OTYPE
```

Use `tt_int_error()`, which returns `Tt_status`, to determine if the `Tt_`address integer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_address_set

```
Tt_status    tt_message_address_set(Tt_message m,
             Tt_address a)
```

Sets the address attribute for the specified message.

Arguments

`Tt_message m`

Opaque handle for the message involved in this operation.

`Tt_address a`

Specifies which message attributes form the address to which the message will be delivered. Possible values are:

```
TT_PROCEDURE
TT_OBJECT
TT_HANDLER
TT_OTYPE
```

Returned Value

`Tt_status`

The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_arg_add

```
Tt_status    tt_message_arg_add(Tt_message m, Tt_mode n,
             const char *vtype, const char *value)
```

Adds a new argument to a message object. Add all arguments before the message is sent.

Note: Do not add arguments to a reply. Only change existing argument values with modes of `TT_OUT` or `TT_INOUT`.

Arguments

`Tt_message m`

Opaque handle for the message involved in this operation.

`Tt_mode n`

Specifies who (sender, handler, observers) writes and reads a message argument. Possible modes are:

```
TT_IN
TT_OUT
TT_INOUT
```

`const char *vtype`

Describes the type of argument data being added. The vtype name is used by the message recipient to interpret data.

`const char *value`

Contents for the message argument attribute. Use `NULL` for values of mode `TT_OUT`, or if the value will be filled in later with `tt_message_arg_val_set`, `tt_message_barg_val_set`, or `tt_message_iarg_val_set`.

Returned Value

`Tt_status`

The status of the operation. Possible values are:

```
TT_OK
TT_ERR_MODE
TT_ERR_NOMP
TT_ERR_POINTER
```

Related Functions

```
tt_message_arg_val_set()
tt_message_barg_add()
tt_message_iarg_add()
```

tt_message_arg_bval

```
Tt_status    tt_message_arg_bval(Tt_message m, int n,
             unsigned char **value, int *len)
```

Retrieves the value of the *n*-th message argument as a byte string.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

`int n`
: Number of the argument you want to retrieve. The first argument is 0.

`unsigned char **value`
: Address of a character pointer that the ToolTalk service should aim to a string containing the contents of the argument.

`int *len`
: Address of an integer that the ToolTalk service should set to the length of the value in bytes.

Returned Values

`Tt_status`
: The status of the operation. Possible values are:

```
TT_ERR_NOMP
TT_ERR_NUM
TT_ERR_POINTER
```

`unsigned char **value`
: Address of a character pointer that the ToolTalk service aimed at a string containing the contents of the argument.

`int *len`
: Address of an integer that the ToolTalk service set to the length of the value in bytes.

tt_message_arg_bval_set

```
Tt_status    tt_message_arg_bval_set(Tt_message m, int n,
             const unsigned char *value, int len)
```

Sets the value and the type of the *n*-th message argument as a byte string. You (the sender) can use `tt_message_arg_bval_set` to fill in opaque data.

Also, changes the value of the *n*-th message argument to a byte string. Used by the handler before replying to the message.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

`int n`
: Number of the argument you want to set. The first argument is 0.

`const unsigned char *value`
: Byte string with the contents for the message argument.

`int len`
: Length of the value in bytes.

Returned Values

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_NUM
TT_ERR_POINTER
```

Related Functions

```
tt_message_barg_add()
tt_message_arg_val_set()
tt_message_iarg_val_set()
```

tt_message_arg_ival

```
Tt_status    tt_message_arg_ival(Tt_message m, int n,
             int *value)
```

Retrieves the value of the *n*-th message argument as an integer.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

`int n`
: Number of the argument you want to retrieve. The first argument is 0.

`int *value`
: Pointer to an integer where the ToolTalk service should store the contents of the argument.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_NUM
TT_ERR_POINTER
```

`int value`
: Value of the *n*-th argument.

tt_message_arg_ival_set

```
Tt_status    tt_message_arg_ival_set(Tt_message m, int n,
             int value)
```

Fills in the *n*-th message argument with an integer value.

Also, changes the value of the *n*-th message argument to an integer.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

`int n`
: Number of the argument you want to set. The first argument is 0.

`int value`
: Contents (in integer form) for the message argument.

Returned Values

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_NUM
TT_ERR_POINTER
```

Related Functions

```
tt_message_arg_ival_add()
tt_message_arg_val_set()
tt_message_barg_val_set()
```

tt_message_arg_mode

Tt_mode tt_message_arg_mode(Tt_message m, int n)

Returns the mode of the *n*-th message argument.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

`int n`
: Number of the argument in which you are interested. The first argument is 0.

Returned Value

`Tt_mode`
: Specifies who (sender, handler, observers) writes and reads a message argument. Possible modes are:

```
TT_IN
TT_OUT
TT_INOUT
```

Use `tt_int_error()`, which returns `Tt_status`, to determine if the `Tt_mode` integer is valid.

Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_NUM
TT_ERR_POINTER
```

tt_message_arg_type

```
char         *tt_message_arg_type(Tt_message m, int n)
```

Retrieves the type of the *n*-th message argument.

Arguments

`Tt_message m`

Opaque handle for the message involved in this operation.

`int n`

Number of the argument in which you are interested. The first argument is 0.

Returned Value

`char *`

Type of the *n*-th message argument. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_NUM
TT_ERR_POINTER
```

tt_message_arg_val

```
char         *tt_message_arg_val(Tt_message m, int n)
```

Returns a pointer to the value (assuming it is a character string) of the *n*-th message argument.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

`int n`
: Number of the argument in which you are interested. The first argument is 0.

Returned Value

`char *`
: Contents for the message argument. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_NUM
TT_ERR_POINTER
```

tt_message_arg_val_set

```
Tt_status    tt_message_arg_val_set(Tt_message m, int n,
             const char *value)
```

Changes the value of the *n*-th message argument. Generally used by the handler before replying to the message.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

`int n`
: Number of the argument you want to change. The first argument is 0.

`const char *value`
: Contents for the message argument.

Returned Values

`Tt_status`

The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_NUM
TT_ERR_POINTER
```

tt_message_args_count

```
int          tt_message_args_count(Tt_message m)
```

Returns the number of arguments in the message.

Arguments

`Tt_message m`

Opaque handle for the message involved in this operation.

Returned Value

`int`

Total number of arguments in the message. Use `tt_int_error()`, which returns `Tt_status`, to determine if the integer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_barg_add

```
Tt_status    tt_message_barg_add(Tt_message m, Tt_mode n,
             const char *vtype, const unsigned char
             *value, int len)
```

Adds an argument to a pattern that may have a value containing imbedded nulls.

Note: Do not add arguments to a reply. Only change existing argument values with modes of `TT_OUT` or `TT_INOUT`.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

`Tt_mode n`
: Specifies who (sender, handler, observers) writes and reads a message argument. Possible modes are:

```
TT_IN
TT_OUT
TT_INOUT
```

`const char *vtype`
: Describes the type of argument data being added. The vtype name is used by the message recipient to interpret data. The ToolTalk service treats the value as an opaque byte string. To pass structured data, your application and the receiving application must encode and decode these opaque byte strings. The most common way of doing this is to use XDR.

`const unsigned char *value`
: Value that the ToolTalk service should fill in.

`int len`
: Length of the value in bytes.

Returned Values

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

Related Functions

```
tt_message_barg_val_set()
tt_message_arg_add()
tt_message_iarg_add()
```

tt_message_callback_add

```
Tt_status    tt_message_callback_add(Tt_message m,
             Tt_message_callback f)
```

Registers a callback function that will be automatically invoked by `tt_message_receive` whenever a reply or other state-change to this message is returned.

`Tt_callback_action` is an enum containing the values `TT_CALLBACK_CONTINUE` and `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`. If the callback returns `TT_CALLBACK_CONTINUE`, other callbacks will be run, and if no callback returns `TT_CALLBACK_PROCESSED`, `tt_message_receive` will return the message.

This behavior can be used to create wrappers for ToolTalk messages. A library routine can construct a request, attach a callback to the message, send the message, and process the reply in the callback. By having the callback return `TT_CALLBACK_PROCESSED`, the message reply will not be returned to the main program, so the message and reply are completely hidden. Note that these callbacks are invoked from `tt_message_receive`, so it's still necessary for programs to arrange for `tt_message_receive` to be called when the file descriptor returned by `tt_fd` becomes active.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

`Tt_message_callback f`
: `Tt_message_callback` is a type definition for a pointer to a function declared like: `Tt_callback_action func(Tt_message m, Tt_pattern p)`. The callback is passed the message in question and the pattern that matched it. The pattern handle will be null if the message didn't match a dynamic pattern (this is usually the case for message callbacks.)

Returned Values

`Tt_status`

The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_class

```
Tt_class          tt_message_class(Tt_message m)
```

Retrieves the class attribute from the specified message.

Arguments

`Tt_message m`

Opaque handle for the message involved in this operation.

Returned Value

`Tt_class`

Indicates whether or not the sender wanted an action to take place after the message is received. Possible values are:

```
TT_NOTICE
TT_REQUEST
```

Use `tt_int_error()`, which returns `Tt_status`, to determine if the `Tt_class` integer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_class_set

```
Tt_status     tt_message_class_set(Tt_message m,
              Tt_class c)
```

Sets the class attribute for the specified message.

Arguments

`Tt_message m`

Opaque handle for the message involved in this operation.

`Tt_class c`

Indicates whether or not you want an action to take place after the message is received. Possible values are:

```
TT_NOTICE
TT_REQUEST
```

Returned Values

`Tt_status`

The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_create

```
Tt_message    tt_message_create(void)
```

Creates a new message object. The ToolTalk service returns a message handle that's really an opaque pointer to a ToolTalk structure. You do not manipulate the structure directly.

Returned Value

`Tt_message`

The unique opaque handle that identifies your message object. If ToolTalk is unable to create a message when requested, an invalid handle will be returned to you. When you attempt to use this handle, the ToolTalk service will report an error. Use `tt_pointer_error` to determine why the ToolTalk service was not able to create the message. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_NUM
```

```
TT_ERR_POINTER
```

Related Functions

```
tt_message_send()
tt_message_destroy()
```

tt_message_create_super

```
Tt_message    tt_message_create_super(Tt_message m)
```

Re-addresses the specified message to the parent otype of the otype or object listed in the message. Returns the re-addressed message so you can fill in additional message attributes and send the message.

Arguments

`Tt_message m`

Opaque handle for the message involved in this operation.

Returned Value

`Tt_message`

Opaque unique handle for the re-addressed message. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_ADDRESS
TT_ERR_NOMP
TT_ERR_OBJID
TT_ERR_OTYPE
TT_ERR_POINTER
```

Related Functions

```
tt_message_send()
tt_message_destroy()
```

tt_message_destroy

```
Tt_status    tt_message_destroy(Tt_message m)
```

Destroys the message. Destroying a message has no effect on the delivery of a message you have already sent.

If you sent a request and are expecting a reply with return values, destroy a message after you have received the reply. If you sent a notice, you can destroy the message after you send it.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

Related Functions

```
tt_message_create()
tt_message_create_super()
```

tt_message_disposition

`Tt_disposition tt_message_disposition(Tt_message m)`

Retrieves the disposition attribute from the specified message.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

Returned Value

`Tt_disposition`

Indicates whether the receiver should be started to receive the message or if the message should be queued until the receiving process is started at a later time. Possible values are:

```
TT_QUEUE
TT_START
TT_QUEUE+TT_START
```

Use `tt_int_error()`, which returns `Tt_status`, to determine if the `Tt_`disposition integer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_disposition_set

```
Tt_status    tt_message_disposition_set(Tt_message m,
             Tt_disposition r)
```

Sets the disposition attribute for the specified message.

Arguments

`Tt_message m`

Opaque handle for the message involved in this operation.

`Tt_disposition r`

Indicates whether the receiver should be started to receive the message or if the message should be queued until the receiving process is started at a later time. Possible values are:

```
TT_QUEUE
TT_START
TT_QUEUE+TT_START
```

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

  `TT_OK`
  `TT_ERR_NOMP`
  `TT_ERR_POINTER`

tt_message_fail

```
Tt_status    tt_message_fail(Tt_message m)
```

Informs the ToolTalk service that your process can not handle the request you just received and that the message should not be offered to other processes of the same ptype as yours. The ToolTalk service will send the message back to the sender with state `TT_FAILED`.

To help the requestor distinguish this case from the case where a message failed because no matching handler could be found, place an explanatory message code in the status attribute of the message with `tt_message_status_set` and `tt_message_status_string_set` before calling `tt_message_fail`.

Note: The status value must be greater than 2047 (`TT_ERR_LAST`) to avoid confusion with the ToolTalk service status values.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

  `TT_OK`
  `TT_ERR_NOMP`
  `TT_ERR_NOTHANDLER`
  `TT_ERR_POINTER`

Related Functions

```
tt_message_status_set()
tt_message_status_string_set()
```

tt_message_file

```
char          *tt_message_file(Tt_message m)
```

Retrieves the file attribute from the specified message.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

Returned Value

`char *`
: File attribute of the specified message. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_file_set

```
Tt_status     tt_message_file_set(Tt_message m,
              const char *file)
```

Sets the file attribute for the specified message.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

Returned Value

`Tt_status`

The status of the operation. Possible values are:

```
TT_OK
TT_ERR_FILE
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_gid

```
gid_t        tt_message_gid(Tt_message m)
```

Retrieves the group ID attribute from the specified message.

The ToolTalk service automatically sets the group ID of a message with the group ID of the process that created the message.

Arguments

`Tt_message m`

Opaque handle for the message involved in this operation.

Returned Value

`gid_t`

The group ID of the message. If the “nobody” group is returned, the message handle is not valid.

Related Functions

`tt_message_uid()`

tt_message_handler

```
char         *tt_message_handler(Tt_message m)
```

Retrieves the handler attribute from the specified message.

Arguments

`Tt_message m`

Opaque handle for the message involved in this operation.

Returned Value

`char *`

Character value that uniquely identifies the process that should handle the message (`Tt_state` = `TT_CREATED` or `TT_SENT`) or the process that did handle the message (`Tt_state` = `TT_SENT` or `TT_HANDLED`).

Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_handler_ptype

```
char         *tt_message_handler_ptype(Tt_message m)
```

Retrieves the handler ptype attribute from the specified message.

Arguments

`Tt_message m`

Opaque handle for the message involved in this operation.

Returned Value

`char *`

Type of process that should handle this message.

Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_handler_ptype_set

```
Tt_status    tt_message_handler_ptype_set(Tt_message m,
             const char *ptid)
```

Sets the handler process type (ptype) attribute for the specified message.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

`const char *ptid`
: Type of process that should or did handle this message.

Returned Value

Tt_status
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_handler_set

```
Tt_status    tt_message_handler_set(Tt_message m,
             const char *procid)
```

Sets the handler attribute for the specified message.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

`const char *procid`
: Character value that uniquely identifies the process you want to handle the message.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:
  `TT_OK`
  `TT_ERR_NOMP`
  `TT_ERR_POINTER`

tt_message_iarg_add

```
Tt_status    tt_message_iarg_add(Tt_messagem, Tt_mode n,
                                 const char *vtype, int value)
```

Adds a new argument to a message object and sets the value to a given integer. Add all arguments before the message is sent.

Note: Do not add arguments to a reply. Only change existing argument values with modes of `TT_OUT` or `TT_INOUT`.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

`Tt_mode n`
: Specifies who (sender, handler, observers) writes and reads a message argument. Possible modes are:
  `TT_IN`
  `TT_OUT`
  `TT_INOUT`

`const char *vtype`
: Describes the type of argument data being added. The vtype name is used by the message recipient to interpret data.

`int value`
: Value to fill in.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_MODE
TT_ERR_NOMP
TT_ERR_POINTER
TT_ERR_VTYPE
```

Related Functions

```
tt_message_arg_ival_set()
tt_message_arg_add()
tt_message_barg_add()
```

tt_message_object

```
char         *tt_message_object(Tt_message m)
```

Retrieves the object attribute from the specified message.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

Returned Value

`char *`
: Object involved in this message. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_OBJID
TT_ERR_POINTER
```

tt_message_object_set

```
Tt_status    tt_message_object_set(Tt_message m,
             const char *objid)
```

Sets the object attribute for the specified message.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

`const char *objid`
: Object involved in this message.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_op

```
char          *tt_message_op(Tt_message m)
```

Retrieves the operation (op) attribute from the specified message.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

Returned Value

`char *`
: Operation the receiver should perform. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_op_set

```
Tt_status     tt_message_op_set(Tt_message m,
              const char *opname)
```

Sets the operation (op) attribute for the specified message.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

`const char *opname`
: Operation the receiver should perform.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_opnum

```
int          tt_message_opnum(Tt_message m)
```

Retrieves the operation number (opnum) attribute from the specified message.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

Returned Value

`int`
: The number of the operation (opnum) involved in this message. Use `tt_int_error()`, which returns `Tt_status`, to determine if the `Tt_`disposition integer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_otype

```
char          *tt_message_otype(Tt_message m)
```

Retrieves the object type (otype) attribute from the specified message.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

Returned Value

`char *`
: Type of the object involved in this message. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_otype_set

```
Tt_status     tt_message_otype_set(Tt_message m,
              const char *otype)
```

Sets the object type (otype) attribute for the specified message.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

`const char *otype`
: Type of the object involved in this message.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
```

```
TT_ERR_NOMP
TT_ERR_OTYPE
TT_ERR_POINTER
```

tt_message_pattern

```
Tt_pattern   tt_message_pattern(Tt_message m)
```

Retrieves the pattern attribute from the specified message.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

Returned Value

`Tt_pattern`
: Opaque handle for a message pattern. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the handle is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_receive

```
Tt_message   tt_message_receive(void)
```

Returns a handle for the next message waiting to be delivered to your process. `tt_message_receive()` also runs any message or pattern callbacks applicable to this message. Check Tt_status with `tt_message_status()` to see if the return value is `TT_WRN_STARTING`. If it is, the ToolTalk service started your application to deliver this message. You must reply to this message.

Note: If the returned handle is 0, no message is available. This can occur if a message or pattern callback processes the message. It can also happen if the time between the `tt_fd()` file descriptor becoming active and the `tt_message_receive()` call is too long. The ToolTalk service will time out and offer the message to another process.

Returned Value

`Tt_message`

Handle for the message object. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the handle is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
```

tt_message_reject

```
Tt_status    tt_message_reject(Tt_message m)
```

Informs the ToolTalk service that your process can not handle this message. The ToolTalk service will try other handlers.

Arguments

`Tt_message m`

Opaque handle for the message involved in this operation.

Returned Value

`Tt_status`

The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_NOTHANDLER
TT_ERR_POINTER
```

tt_message_reply

```
Tt_status    tt_message_reply(Tt_message m)
```

Informs the ToolTalk service that your process has finished handling the message, and all return values (any arguments with the TT_OUT or TT_INOUT mode) have been filled in. The ToolTalk service will send the message back to the sender and fill the state attribute in with TT_HANDLED.

Arguments

`Tt_message m`

Opaque handle for the message involved in this operation.

Returned Value

`Tt_status`

The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_NOTHANDLER
TT_ERR_POINTER
TT_ERR_PROCID
```

tt_message_scope

```
Tt_scope     tt_message_scope(Tt_message m)
```

Retrieves the scope attribute from the specified message.

Arguments

`Tt_message m`

Opaque handle for the message involved in this operation.

Returned Value

`Tt_scope`

Identifies the set of processes eligible to receive the message. Possible values are:

```
TT_SESSION
TT_FILE
TT_BOTH
TT_FILE_IN_SESSION
```

Use `tt_int_error()`, which returns `Tt_status`, to determine if the `Tt_`scope integer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_scope_set

```
Tt_status    tt_message_scope_set(Tt_message m,
             Tt_scope s)
```

Sets the scope attribute for the specified message.

Arguments

`Tt_message m`

Opaque handle for the message involved in this operation.

`Tt_scope s`

Identifies the set of processes eligible to receive the message. Possible values are:

```
TT_SESSION
TT_FILE
TT_BOTH
TT_FILE_IN_SESSION
```

Returned Value

`Tt_status`

The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_send

```
Tt_status    tt_message_send(Tt_message m)
```

Sends the specified message.

Arguments

`Tt_message m`

Opaque handle for the message involved in this operation.

Returned Value

`Tt_status`

The status of the operation. Possible values are:

```
TT_OK
TT_ERR_ADDRESS
TT_ERR_CLASS
TT_ERR_FILE
TT_ERR_NOMP
TT_ERR_OBJID
TT_ERR_OTYPE
TT_ERR_OVERFLOW
TT_ERR_POINTER
TT_ERR_PROCID
TT_ERR_SESSION
TT_WRN_STALE_OBJID
```

tt_message_sender

```
char          *tt_message_sender(Tt_message m)
```

Retrieves the sender attribute from the specified message.

Arguments

`Tt_message m`

Opaque handle for the message involved in this operation.

Returned Value

`char *`

Character value that uniquely identifies the process that sent the message. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
```

```
TT_ERR_POINTER
```

tt_message_sender_ptype

```
char          *tt_message_sender_ptype(Tt_message m)
```

Retrieves the sender ptype attribute from the specified message.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

Returned Value

`char *`
: Process that sent this message. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_sender_ptype_set

```
Tt_status     tt_message_sender_ptype_set(Tt_message m,
              const char *ptid)
```

Sets the sender ptype attribute for the specified message.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

`const char *ptid`
: Type of process that is sending this message.

Returned Value

`Tt_status`

The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_session

```
char         *tt_message_session(Tt_message m)
```

Retrieves the session attribute from the specified message.

Arguments

`Tt_message m`

Opaque handle for the message involved in this operation.

Returned Value

`char *`

Identifier of the session to which this message applies. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_session_set

```
Tt_status    tt_message_session_set(Tt_message m,
             const char *sessid)
```

Sets the session attribute for the specified message.

Arguments

`Tt_message m`

Opaque handle for the message involved in this operation.

`const char *sessid`
: Identifier of the session in which you are interested.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:
  `TT_OK`
  `TT_ERR_NOMP`
  `TT_ERR_POINTER`

tt_message_state

`Tt_state` `tt_message_state(Tt_message m)`

Retrieves the state attribute from the specified message.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

Returned Value

`Tt_state`
: Indicates a message's current delivery state. Possible values are:

  `TT_CREATED`
  `TT_SENT`
  `TT_HANDLED`
  `TT_FAILED`
  `TT_QUEUED`
  `TT_STARTED`
  `TT_REJECTED`

  Use `tt_int_error()`, which returns `Tt_status`, to determine if the `Tt_state` integer is valid.

  Possible `Tt_status` values are:

  `TT_OK`
  `TT_ERR_NOMP`
  `TT_ERR_POINTER`

tt_message_status

```
int          tt_message_status(Tt_message m)
```

Retrieves the status attribute from the specified message.

Arguments

`Tt_message m`

Opaque handle for the message involved in this operation.

Returned Value

`int`

An integer that describes the status stored in the status attribute of this message.

Use `tt_int_error()`, which returns `Tt_status`, to determine if the integer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

Related Functions

`tt_message_status_string()`

tt_message_status_set

```
Tt_status    tt_message_status_set(Tt_message m,
             int status)
```

Sets the status attribute for the specified message.

Note: The status value must be greater than 2047 (`TT_ERR_LAST`) to avoid confusion with the ToolTalk service status values.

Arguments

`Tt_message m`

Opaque handle for the message involved in this operation.

`int status`
: Status to be stored in this message.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_status_string

```
char         *tt_message_status_string(Tt_message m)
```

Retrieves the character string stored with the status attribute for the specified message.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

Returned Value

`char *`
: Status string stored in this message. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid.

  Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

Related Functions

`tt_message_status()`

tt_message_status_string_set

```
Tt_status    tt_message_status_string_set(Tt_message m,
             const char *status_str)
```

Sets a character string with the status attribute for the specified message.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

`const char *status_str`
: Status string stored in this message.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

Related Functions

`tt_message_status_set()`

tt_message_uid

```
uid_t        tt_message_uid(Tt_message m)
```

Retrieves the user ID attribute from the specified message.

The ToolTalk service automatically sets the user ID of a message with the user ID of the process that created the message.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

Returned Value

`uid_t`

The user ID of the message, or the "nobody" user (65534) if the message handle is not valid.

Related Functions

`tt_message_gid()`

tt_message_user

```
void         *tt_message_user(Tt_message m, int key)
```

Retrieves the user information stored in data cells associated with the specified message object you created. Since the user data is part of the message object (the storage buffer in your application), not the actual message, you can only retrieve user information that you placed on the message.

Arguments

`Tt_message m`

Opaque handle for the message involved in this operation.

`int key`

User data cell in which you are interested. It must be unique over all user data cells for this message.

Returned Value

`void *`

A piece of arbitrary user data that is one word in size. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_message_user_set

```
Tt_status    tt_message_user_set(Tt_message m, int key,
             void *v)
```

Stores user information in data cells associated with the specified message object.

Note that the user data is part of the message object (the storage buffer in your application), not the actual message. Data stored by the sender in user data cells is not seen by any handlers or observers. Use arguments for data that handlers or observers need to see.

Arguments

`Tt_message m`
: Opaque handle for the message involved in this operation.

`int key`
: User data cell in which you are interested.

`void *v`
: A piece of arbitrary user data that is one word in size.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
TT_ERR_PROCID
```

Related Functions

`tt_message_arg_add()`

tt_objid_equal

```
int          tt_objid_equal(const char *objid1,
             const char *objid2)
```

Tests to see if two objids are equal. `tt_objid_equal()` is better than `strcmp` for the purpose since it returns "1" even in the case where one objid is a forwarding pointer for the other.

Arguments

`const char *objid1`
: Identifier of one of the objects involved in this operation.

`const char *objid2`Identifier of the other object involved in this operation.

Returned Value

`int`
: Integer indicating whether or not the objids are equal. Possible values are:

```
0 – no
1 – yes
```

Use `tt_int_error()`, which returns `Tt_status`, to determine if the integer is valid.

Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_OBJID
```

tt_objid_objkey

```
char         *tt_objid_objkey(const char *objid)
```

Returns the "unique key" portion of a objid.

Arguments

`const char *objid`
: Identifier of the object involved in this operation.

Returned Value

`char *`

Unique key of the objid. No two objids have the same unique key.

Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_OBJID
```

tt_onotice_create

```
Tt_message   tt_onotice_create(const char *objid,
             const char *op)
```

Creates a message with:

`Tt_address = TT_OBJECT`

`Tt_class = TT_NOTICE`

The handle for the created message is returned so you can add arguments, other attributes, and send the message.

Arguments

`const char *objid`

Identifier of the desired object.

`const char *op`

Operation to be performed by the receiver.

Returned Value

`Tt_message`

The unique handle that identifies your message.

Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the handle is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
```

```
TT_ERR_PROCID
```

tt_open

```
char          *tt_open(void)
```

Returns the process identifier (procid) for the calling process, and sets this procid as the default procid for the process. `tt_open()` is typically the first the ToolTalk function you call from your process. A process may call `tt_open()` more than once to obtain more than one procid. Each procid has its own associated `tt_fd()` file descriptor, and can join another session. To switch to another procid use `tt_default_procid_set()`.

Returned Value

`char *`

Character value that uniquely identifies your process.

Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
```

Related Functions

```
tt_fd()
tt_default_procid_set()
```

tt_orequest_create

```
Tt_message    tt_orequest_create(const char *objid,
              const char *op)
```

Creates a message with:

Tt_address = TT_OBJECT
`Tt_class = TT_REQUEST`

The handle for the created message is returned so you can add arguments, other attributes, and send the message.

Arguments

`const char *objid`
: Identifier of the desired object.

`const char *op`
: The operation to be performed by the receiver.

Returned Value

`Tt_message`
: The unique handle that identifies your message. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the handle is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_PROCID
```

tt_otype_base

```
char         *tt_otype_base(const char *otype)
```

Returns the base otype that the given otype is derived from, or NULL if the given otype is not derived.

Arguments

`char *otype`
: Object type involved in this operation.

Returned Value

`char *`
: Name of the base otype, or NULL if the given otype is not derived. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_OTYPE
```

Related Functions

```
tt_otype_is_derived()
tt_otype_derived()
tt_otype_deriveds_count()
tt_spec_type()
tt_message_otype()
```

tt_otype_derived

```
char         *tt_otype_derived(const char *otype, int i)
```

Returns the *i*-th otype derived from the given otype.

Arguments

`const char *otype`
: Object type involved in this operation.

`int i`
: Zero-based index into the otypes derived from the given otype.

Returned Value

`char *`
: Name of the *i*-th otype derived from the given otype. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_OTYPE
```

Related Functions

```
tt_otype_is_derived()
tt_otype_base()
tt_otype_deriveds_count()
tt_spec_type()
tt_message_otype()
```

tt_otype_deriveds_count

```
int          tt_otype_deriveds_count(const char *otype)
```

Returns the number of otypes derived from the given otype.

Arguments

`const char *otype`
: Object type involved in this operation.

Returned Value

`int`
: The number of otypes derived from the given otype. Use `tt_int_error()`, which returns `Tt_status`, to determine if the integer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_OTYPE
```

Related Functions

```
tt_otype_is_derived()
tt_otype_base()
tt_otype_derived()
tt_spec_type()
tt_message_otype()
```

tt_otype_hsig_arg_mode

```
Tt_mode      tt_otype_hsig_arg_mode(const char *otype,
             int sig, int arg)
```

Returns the Tt_mode of the arg'th argument of the sig'th request signature of the given otype.

Arguments

`const char *otype`
: Object type involved in this operation.

`int sig`

Zero-based index into the request signatures of the specified otype.

`int arg`

Zero-based index into the arguments of the specified signature.

Returned Value

`Tt_mode`

The `Tt_mode` of the specified argument, which determines who (sender or handler) writes and reads a message argument. Possible modes are:

```
TT_IN
TT_OUT
TT_INOUT
```

Use `tt_int_error()`, which returns `Tt_status`, to determine if the integer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_NUM
TT_ERR_OTYPE
```

Related Functions

```
tt_otype_hsig_arg_type()
tt_otype_hsig_count()
tt_otype_hsig_args_count()
tt_otype_hsig_op()
```

tt_otype_hsig_arg_type

```
char         *tt_otype_hsig_arg_type(const char *otype,
             int sig, int arg)
```

Returns the data type of the arg'th argument of the sig'th request signature of the given otype.

Arguments

`const char *otype`
: Object type involved in this operation.

`int sig`
: Zero-based index into the request signatures of the specified otype.

`int arg`
: Zero-based index into the arguments of the specified signature.

Returned Value

`char *`
: Data type of the specified argument. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_NUM
TT_ERR_OTYPE
```

Related Functions

```
tt_otype_hsig_arg_mode()
tt_otype_hsig_count()
tt_otype_hsig_args_count()
tt_otype_hsig_op()
```

tt_otype_hsig_args_count

```
int          tt_otype_hsig_args_count(const char *otype,
             int sig)
```

Returns the number of arguments of the sig'th request signature of the given otype.

Arguments

`const char *otype`
: Object type involved in this operation.

`int sig`

Zero-based index into the request signatures of the specified otype.

Returned Value

`int`

The number of arguments of the sig'th request signature of the given otype. Use `tt_int_error()`, which returns `Tt_status`, to determine if the integer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_NUM
TT_ERR_OTYPE
```

Related Functions

```
tt_otype_hsig_arg_type()
tt_otype_hsig_arg_mode()
tt_otype_hsig_count()
tt_otype_hsig_op()
```

tt_otype_hsig_count

```
int          tt_otype_hsig_count(const char *otype)
```

Returns the number of request signatures for the given otype.

Arguments

`const char *otype`

Object type involved in this operation.

Returned Value

`int`

The number of request signatures for the given otype. Use `tt_int_error()`, which returns `Tt_status`, to determine if the integer is valid. Possible `Tt_status` values are:

```
TT_OK
```

```
TT_ERR_NOMP
TT_ERR_OTYPE
```

Related Functions

```
tt_otype_hsig_arg_type()
tt_otype_hsig_arg_mode()
tt_otype_hsig_args_count()
tt_otype_hsig_op()
```

tt_otype_hsig_op

char *tt_otype_hsig_op(const char *otype, int sig)

Returns the op name of the sig'th request signature of the give otype.

Arguments

`const char *otype`
: Object type involved in this operation.

`int sig`
: Zero-based index into the request signatures of the given otype.

Returned Value

`char *`
: Operation attribute of the specified request signature. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_NUM
TT_ERR_OTYPE
```

Related Functions

```
tt_otype_hsig_arg_type()
tt_otype_hsig_arg_mode()
tt_otype_hsig_args_count()
tt_otype_hsig_count()
```

tt_otype_is_derived

```
int          tt_otype_is_derived(const char *derivedotype,
             const char *baseotype)
```

Returns 1 if and only if derivedotype is derived directly or indirectly from baseotype.

Arguments

`const char *derivedotype`
: The purportedly derived otype.

`const char *baseotype`
: Candidate base otype.

Returned Value

`int`
: Returns 1 if and only if derivedotype is derived directly or indirectly from baseotype. Use `tt_int_error()`, which returns `Tt_status`, to determine if the integer is valid. Possible `Tt_status` values are:

  ```
  TT_OK
  TT_ERR_NOMP
  TT_ERR_OTYPE
  ```

Related Functions

```
tt_otype_deriveds_count()
tt_otype_base()
tt_otype_derived()
tt_spec_type()
tt_message_otype()
```

tt_otype_osig_arg_mode

```
Tt_mode      tt_otype_osig_arg_mode(const char *otype, int
             sig, int arg)
```

Returns the Tt_mode of the arg'th argument of the sig'th notice signature of the given otype.

Arguments

`const char *otype`
: Object type involved in this operation.

`int sig`
: Zero-based index into the notice signatures of the specified otype.

`int arg`
: Zero-based index into the arguments of the specified signature.

Returned Value

`Tt_mode`
: The `Tt_mode` of the specified argument, which determines who (sender or handler) writes and reads a message argument. Possible modes are:

```
TT_IN
TT_OUT
TT_INOUT
```

Use `tt_int_error()`, which returns `Tt_status`, to determine if the `Tt_mode` value is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_NUM
TT_ERR_OTYPE
```

Related Functions

```
tt_otype_osig_arg_type()
tt_otype_osig_count()
tt_otype_osig_args_count()
tt_otype_osig_op()
```

tt_otype_osig_arg_type

```
char         *tt_otype_osig_arg_type(const char *otype, int
             sig, int arg)
```

Returns the data type of the arg'th argument of the sig'th notice signature of the given otype.

Arguments

`const char *otype`
: Object type involved in this operation.

`int sig`
: Zero-based index into the notice signatures of the specified otype.

`int arg`
: Zero-based index into the arguments of the specified signature.

Returned Value

`char *`
: Data type of the specified argument. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_NUM
TT_ERR_OTYPE
```

Related Functions

```
tt_otype_osig_arg_mode()
tt_otype_osig_count()
tt_otype_osig_args_count()
tt_otype_osig_op()
```

tt_otype_osig_args_count

```
int          tt_otype_osig_args_count(const char *otype,
             int sig)
```

Returns the number of arguments of the sig'th notice signature of the given otype.

Arguments

`const char *otype`
: Object type involved in this operation.

`int sig`
: Zero-based index into the notice signatures of the specified otype.

Returned Value

`int`
: The number of arguments of the sig'th notice signature of the given otype. Use `tt_int_error()`, which returns `Tt_status`, to determine if the integer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_NUM
TT_ERR_OTYPE
```

Related Functions

```
tt_otype_osig_arg_type()
tt_otype_osig_arg_mode()
tt_otype_osig_count()
tt_otype_osig_op()
```

tt_otype_osig_count

```
int          tt_otype_osig_count(const char *otype)
```

Returns the number of notice signatures for the given otype.

Arguments

`const char *otype`
: Object type involved in this operation.

Returned Value

`int`

The number of notice signatures for the given otype. Use `tt_int_error()`, which returns `Tt_status`, to determine if the integer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_OTYPE
```

Related Functions

```
tt_otype_osig_arg_type()
tt_otype_osig_arg_mode()
tt_otype_osig_args_count()
tt_otype_osig_op()
```

tt_otype_osig_op

```
char         *tt_otype_osig_op(const char *otype, int sig)
```

Returns the op name of the sig'th notice signature of the give otype.

Arguments

`const char *otype`

Object type involved in this operation.

`int sig`

Zero-based index into the notice signatures of the given otype.

Returned Value

`char *`

Operation attribute of the specified notice signature.

Use `tt_ptr_error()`, which returns `Tt_status`, to determine if pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
```

TT_ERR_NUM
TT_ERR_OTYPE

Related Functions

tt_otype_osig_arg_type()
tt_otype_osig_arg_mode()
tt_otype_osig_args_count()
tt_otype_osig_count()

tt_pattern_address_add

```
Tt_status    tt_pattern_address_add(Tt_pattern p,
             Tt_address d)
```

Adds a value to the address field for the specified pattern.

Arguments

Tt_pattern p

A unique handle for a message pattern. You receive this handle after you issue tt_pattern_create().

Tt_address d

Specifies which pattern attributes form the address that messages will be matched against. Possible values are:

TT_PROCEDURE
TT_OBJECT
TT_HANDLER
TT_OTYPE

Returned Value

Tt_status

The status of the operation. Possible values are:

TT_OK
TT_ERR_NOMP
TT_ERR_POINTER

tt_pattern_arg_add

```
Tt_status    tt_pattern_arg_add(Tt_pattern p, Tt_mode n,
             const char *vtype, const char *value)
```

Adds an argument to a pattern. Add pattern arguments before registering your pattern with the ToolTalk service.

Arguments

`Tt_pattern p`
: Opaque handle for the pattern involved in this operation.

Tt_mode n
: Specifies who (sender, handler, observers) writes and reads a message argument. Possible modes are:

```
TT_IN
TT_OUT
TT_INOUT
```

`const char *vtype`
: Describes the type of argument data being added. The vtype name is used by the message recipient to interpret data. Use 'ALL' to match without regard to argument value type.

`const char *value`
: Value to fill in (must be an unsigned character string.) Use `NULL` to indicate that any value matches.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

Related Functions

```
tt_pattern_register()
tt_pattern_barg_add()
tt_pattern_iarg_add()
```

tt_pattern_barg_add

```
Tt_status    tt_pattern_barg_add(Tt_pattern m, Tt_mode n,
             const char *vtype, const unsigned
             char *value, int len)
```

Adds an argument with a value containing imbedded nulls to a pattern.

Arguments

`Tt_pattern m`
: Opaque handle for the pattern involved in this operation.

`Tt_mode n`
: Specifies who (sender, handler, observers) writes and reads a message argument. Possible modes are:

```
TT_IN
TT_OUT
TT_INOUT
```

`const char *vtype`
: Describes the type of argument data being added. The vtype name is used by the message recipient to interpret data. The ToolTalk service treats the value as an opaque byte string. To pass structured data, your application and the receiving application must encode and decode these unique values. The most common way of doing this is to use XDR.

`const unsigned char *value`
: Value to be filled in. Use `NULL` to specify that any value matches.

`int len`
: Length of the value in bytes.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

Related Functions

```
tt_pattern_register()
tt_pattern_arg_add()
tt_pattern_iarg_add()
```

tt_pattern_callback_add

```
Tt_status     tt_pattern_callback_add(Tt_pattern m,
              Tt_message_callback f)
```

Registers a callback function that will be automatically invoked by `tt_message_receive()` whenever a message matches the pattern.

`Tt_callback_action` is an `enum` containing the values `TT_CALLBACK_CONTINUE` and `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()`; if the callback returns `TT_CALLBACK_CONTINUE`, other callbacks will be run, and if no callback returns `TT_CALLBACK_PROCESSED`, `tt_message_receive()` will return the message.

Arguments

`Tt_pattern m`
: Opaque handle for the pattern involved in this operation.

`Tt_message_callback f`
: `Tt_message_callback` is a type definition for a pointer to a function declared like: `Tt_callback_action func(Tt_message m,Tt_pattern p)`. The callback is passed the message in question and the pattern that matched it.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

Related Functions

`tt_pattern_register()`

tt_pattern_category

```
Tt_category  tt_pattern_category(Tt_pattern p)
```

Returns the category value of the specified pattern.

Arguments

`Tt_pattern p`
: Opaque handle for a message pattern.

Returned Value

`Tt_category`
: Indicates the receiver's intent. Possible values are:

```
TT_OBSERVE
TT_HANDLE
```

Use `tt_int_error()`, which returns `Tt_status`, to determine if the `Tt_category` integer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

Related Functions

`tt_pattern_category_set()`

tt_pattern_category_set

```
Tt_status    tt_pattern_category_set(Tt_pattern p,
             Tt_category c)
```

Fills in the category field for the specified pattern.

Arguments

`Tt_pattern p`

A unique handle for a message pattern. You receive this handle after you issue `tt_pattern_create()`.

`Tt_category c`

Indicates the receiver's intent. Possible values are:

```
TT_OBSERVE
TT_HANDLE
```

Returned Value

`Tt_status`

The status of the operation. Possible values are:

```
TT_OK
TT_ERR_CATEGORY
TT_ERR_NOMP
TT_ERR_POINTER
```

Related Functions

`tt_pattern_category()`

tt_pattern_class_add

```
Tt_status    t_pattern_class_add(Tt_pattern p,
             Tt_class c)
```

Adds a value to the class information for the specified pattern. If the class is `TT_REQUEST`, the sender expects a reply to the message. If the class is `TT_NOTICE`, the sender will not expect a reply.

Arguments

`Tt_pattern p`

A unique handle for a message pattern. You receive this handle after you issue `tt_pattern_create()`.

Tt_class c

Indicates whether or not the sender wants the receiver to take action after the message is received. Possible values are:

```
TT_NOTICE
TT_REQUEST
```

`Tt_pattern p`

A unique handle for a message pattern. You receive this handle after you issue `tt_pattern_create()`.

`Tt_class c`

Indicates whether or not the sender wants the receiver to take action after the message is received. Possible values are:

```
TT_NOTICE
TT_REQUEST
```

Returned Value

`Tt_status`

The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_pattern_create

```
Tt_pattern   tt_pattern_create(void)
```

Requests a new pattern object. After receiving the pattern object, fill in the message pattern fields to indicate what type of messages you want to receive and register it with the ToolTalk service.

Note: You can supply multiple values for each attribute you *add* to a pattern (some attributes are *set* and only have one value). 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.

Returned Value

`Tt_pattern`

Opaque handle for a message pattern. Use this handle in future calls to identify the pattern object. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
```

Related Functions

`tt_pattern_register()`

tt_pattern_destroy

```
Tt_status    tt_pattern_destroy(Tt_pattern p)
```

Destroys a pattern object. Destroying a pattern object automatically unregisters the pattern with the ToolTalk service.

Arguments

`Tt_pattern p`

A unique handle for a message pattern. You receive this handle after you issue `tt_pattern_create()`.

Returned Value

`Tt_status`

The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

Related Functions

`tt_pattern_register()`

tt_pattern_disposition_add

```
Tt_status    tt_pattern_disposition_add(Tt_pattern p,
             tt_disposition r)
```

Adds a value to the disposition field for the specified pattern.

Arguments

`Tt_pattern p`
: A unique handle for a message pattern. You receive this handle after you issue `tt_pattern_create()`.

`Tt_disposition r`
: Indicates whether the receiver 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 thrown away if the receiver is not started. Possible values are:

```
TT_DISCARD
TT_QUEUE
TT_START
TT_QUEUE+TT_START
```

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_pattern_file_add

```
Tt_status    tt_pattern_file_add(Tt_pattern p,
             const char *file)
```

Adds a value to the file field of the specified pattern.

Note: When you join a file, the ToolTalk service updates the file field of your registered patterns.

Arguments

`Tt_pattern p`
: A unique handle for a message pattern. You receive this handle after you issue `tt_pattern_create()`.

`const char *file`
: Name of the file in which you are interested.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_pattern_arg_add

```
Tt_status    tt_pattern_iarg_add(Tt_pattern m, Tt_mode n,
             const char *vtype, int value)
```

Adds a new argument to a pattern and sets the value to a given integer. Add all arguments before the pattern is registered.

Arguments

`Tt_pattern m`
: Opaque handle for the pattern involved in this operation.

`Tt_mode n`
: Specifies who (sender, handler, observers) writes and reads a message argument. Possible modes are:

```
TT_IN
TT_OUT
TT_INOUT
```

`const char *vtype`
: Describes the type of argument data being added. The vtype name is used by the message recipient to interpret data.

`int value`
: Value to fill in.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_MODE
TT_ERR_NOMP
TT_ERR_POINTER
TT_ERR_VTYPE
```

Related Functions

`tt_pattern_register()`

tt_pattern_object_add

```
Tt_status    tt_pattern_object_add(Tt_pattern p,
             const char *objid)
```

Adds a value to the object field of the specified pattern.

Arguments

`Tt_pattern p`
: A unique handle for a message pattern. You receive this handle after you issue `tt_pattern_create()`.

`const char *objid`
: Identifier for the specified object. Objid's are returned from `tt_spec_create()` or `tt_spec_move()`.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_pattern_op_add

```
Tt_status    tt_pattern_op_add(Tt_pattern p,
             const char *opname)
```

Adds a value to the operation field of the specified pattern.

Arguments

`Tt_pattern p`
: A unique handle for a message pattern. You receive this handle after you issue `tt_pattern_create()`.

`const char *opname`
: The name of the operation (op) your process can perform.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_pattern_otype_add

```
Tt_status    tt_pattern_otype_add(Tt_pattern p,
             const char *otype)
```

Adds a value to the object type (otype) field for the specified pattern.

Arguments

`Tt_pattern p`
: A unique handle for a message pattern. You receive this handle after you issue `tt_pattern_create()`.

`const char *otype`
: The name of the object type your application manages.

Returned Value

`Tt_status`

The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_OTYPE
TT_ERR_POINTER
```

tt_pattern_register

```
Tt_status    tt_pattern_register(Tt_pattern p)
```

Registers your pattern with ToolTalk, so that your process will start receiving messages that match the pattern. Once a pattern is registered, no further changes can be made in the pattern.

Note: When you join a session or file, the ToolTalk service updates the file and session field of your registered patterns.

Arguments

`Tt_pattern p`

A unique handle for a message pattern. You receive this handle after you issue `tt_pattern_create()`.

Returned Value

`Tt_status`

The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
TT_ERR_PROCID
```

Related Functions

`tt_pattern_unregister()`

tt_pattern_scope_add

```
Tt_status    tt_pattern_scope_add(Tt_pattern p,
             Tt_scope s)
```

Adds a value to the scope field for the specified pattern.

Arguments

`Tt_pattern p`

A unique handle for a message pattern. You receive this handle after you issue `tt_pattern_create()`.

`Tt_scope s`

Specifies what process are eligible to receive the message. Possible values are:

```
TT_SESSION
TT_FILE
TT_BOTH
TT_FILE_IN_SESSION
```

Returned Value

`Tt_status`

The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_pattern_sender_add

```
Tt_status    tt_pattern_sender_add(Tt_pattern p,
             const char *procid)
```

Adds a value to the sender field for the specified pattern.

Arguments

`Tt_pattern p`

A unique handle for a message pattern. You receive this handle after you issue `tt_pattern_create()`.

`const char *procid`
: Character value that uniquely identifies the process in which you are interested.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

  `TT_OK`
  `TT_ERR_NOMP`
  `TT_ERR_POINTER`

tt_pattern_sender_ptype_add

```
Tt_status    tt_pattern_sender_ptype_add(Tt_pattern p,
             const char *ptid)
```

Adds a value to the sender's process type (ptype) field for the specified pattern.

Arguments

`Tt_pattern p`
: A unique handle for a message pattern. You receive this handle after you issue `tt_pattern_create()`.

`const char *ptid`
: Use the character string that uniquely identifies the type of process in which you are interested.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

  `TT_OK`
  `TT_ERR_NOMP`
  `TT_ERR_POINTER`

tt_pattern_session_add

```
Tt_status    tt_pattern_session_add(Tt_pattern p,
             const char *sessid)
```

Adds a value to the session field for the specified pattern.

Note: When you join a session, the ToolTalk service updates the session field of your registered patterns.

Arguments

`Tt_pattern p`
: A unique handle for a message pattern. You receive this handle after you issue `tt_pattern_create()`.

`const char *sessid`
: Session in which you are interested.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

  ```
  TT_OK
  TT_ERR_NOMP
  TT_ERR_POINTER
  ```

tt_pattern_state_add

```
Tt_status    tt_pattern_state_add(Tt_pattern p, Tt_state s)
```

Adds a value to the state field for the specified pattern.

Arguments

`Tt_pattern p`
: A unique handle for a message pattern. You receive this handle after you issue `tt_pattern_create()`

  ```
  TT_CREATED
  TT_SENT
  TT_HANDLED
  ```

```
TT_FAILED
TT_QUEUED
TT_STARTED
TT_REJECTED
```

Returned Value

`Tt_status`

The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_pattern_unregister

```
Tt_status    tt_pattern_unregister(Tt_pattern p)
```

Unregisters the specified pattern from the ToolTalk service. Your process will stop receiving messages that match this pattern.

Arguments

`Tt_pattern p`

A unique handle for a message pattern. You receive this handle after you issue `tt_pattern_create()`.

Returned Value

`Tt_status`

The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

Related Functions

`tt_pattern_register()`

tt_pattern_user

```
void         *tt_pattern_user(Tt_pattern p, int key)
```

Returns the value in the indicated user data cell for the specified pattern object.

Every pattern object allows an arbitrary number of user data cells, each one word (a `void` *) in size. The user data cells are identified by integer keys. Your tool can use these in any way you see fit, to associate arbitrary data with a pattern object. Note that the user data is part of the pattern object (the storage buffer in your application), not the actual pattern. The content of user cells has no effect on pattern matching.

Arguments

`Tt_pattern p`

A unique handle for a message pattern. You receive this handle after you issue `tt_pattern_create()`.

`int key`

User data cell in which you are interested. Your application assigns the keys to the user data cells which are part of the pattern object with `tt_pattern_user_set()`. Values must be unique over all data cells for this pattern.

Returned Value

`void *`

A piece of arbitrary user data that is one word in size. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

Related Functions

`tt_pattern_user_set()`

tt_pattern_user_set

```
Tt_status    tt_pattern_user_set(Tt_pattern p, int key,
             void *v)
```

Stores information in the user data cells associated with the specified pattern object.

Arguments

Tt_pattern p
: A unique handle for a message pattern. You receive this handle after you issue `tt_pattern_create()`.

`int key`
: User data cell in which you are interested. Values must be unique over all data cells for this pattern.

`void *v`
: A piece of arbitrary user data that is one word in size.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

Related Functions

`tt_pattern_user()`

tt_pnotice_create

```
Tt_message   tt_pnotice_create(Tt_scope scope,
             const char *op)
```

Creates a message with:

`Tt_address = TT_PROCEDURE`

`Tt_class = TT_NOTICE`

The handle for the created message is returned so you can add arguments, other attributes, and send the message.

Arguments

`Tt_scope scope`

A portion of the message that helps determine which processes are eligible to receive the message. A potential recipient could be joined to:

```
TT_SESSION
TT_FILE
TT_BOTH
TT_FILE_IN_SESSION
```

If the scope is `TT_SESSION`, the session is set to the current default session. If the scope is `TT_FILE`, the file is set to the current default file. If the scope is `BOTH` or `FILE_IN_SESSION`, both file and session are set to the defaults.

`const char *op`

The operation to be performed by the receiver.

Returned Value

`Tt_message`

The unique handle that identifies your message. If ToolTalk is unable to create a message when requested, an invalid handle will be returned to you. When you attempt to use this handle, the ToolTalk service will report an error. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_PROCID
```

tt_pointer_error

```
Tt_status    tt_pointer_error(void *pointer)
```

Given an opaque pointer (`Tt_message` or `Tt_pattern`), or character pointer (`char *`), returns `TT_OK` if the pointer is valid or the encoded `Tt_status` value if the pointer is an error object.

To avoid the annoyance of having to cast the opaque or character pointer to `void *` in every call, a macro `tt_ptr_error(p)` is provided that expands to `tt_pointer_error((void *)(p))`.

Arguments

`void *pointer`
: Opaque pointer or character pointer to be checked.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_POINTER
```

tt_prequest_create

```
Tt_message   tt_prequest_create(Tt_scope scope,
              const char *op)
```

Creates a message with:

```
Tt_address = TT_PROCEDURE
Tt_class = TT_REQUEST
```

The handle for the created message is returned so you can add arguments, other attributes, and send the message.

Arguments

`Tt_scope scope`
: A portion of the message that helps determine which processes are eligible to receive the message. A potential recipient could be joined to:

```
TT_SESSION
```

```
TT_FILE
TT_BOTH
TT_FILE_IN_SESSION
```

If the scope is `TT_SESSION`, the session is set to the current default session. If the scope is `TT_FILE`, the file is set to the current default file. If the scope is `BOTH` or `FILE_IN_SESSION`, both file and session are set to the defaults.

`const char *op`
The operation to be performed by the receiver.

Returned Value

`Tt_message`
The unique handle that identifies your message. If ToolTalk is unable to create a message when requested, an invalid handle will be returned to you. When you attempt to use this handle, the ToolTalk service will report an error. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_PROCID
```

tt_ptr_error

```
Tt_status    tt_ptr_error(pointer)
```

A macro that expands to `tt_pointer_error((void *)(p))`. `tt_ptr_error()` helps you avoid the annoyance of having to cast the opaque or character pointer to `void *` in every call.

Arguments

`pointer`
Opaque pointer or character pointer to be checked.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:
  `TT_OK`
  `TT_ERR_NOMP`
  `TT_ERR_POINTER`

tt_ptype_declare

```
Tt_status    tt_ptype_declare(const char *ptid)
```

Registers your process type (ptype) with the ToolTalk service.

Arguments

`const char *ptid`
: Use the character string specified in your ptype that uniquely identifies your process.

Returned Value

`Tt_status`
: The status of the operation. Possible values are:
  `TT_OK`
  `TT_ERR_NOMP`
  `TT_ERR_PTYPE`

tt_release

```
void         tt_release(int mark)
```

Frees all storage allocated on the ToolTalk API allocation stack since your `tt_mark()` call.

Your application typically calls this at the end of a procedure to release all storage allocated within the procedure.

Arguments

`int mark`
: Integer that marks your application's storage position in the ToolTalk API allocation stack.

Related Functions

`tt_mark()`

tt_session_bprop

```
Tt_status    tt_session_bprop(const char *sessid,
             const char *propname, int i,
             unsigned char **value, int *length)
```

Obtains the *i*-th value (first value is number 0) of the named property of the session identified by sessid. If there are *i* values or fewer, both returned value and returned length are zeroed.

Arguments

`const char *sessid`
: The session you have joined. Use the sessid value the ToolTalk service returns after you issue `tt_default_session()`.

`const char *propname`
: The name of the property from which you want to obtain values.

`int i`
: The number of the item in the property list for which you want to obtain the value. The list numbering begins with 0.

`unsigned char **value`
: Address of a character pointer that the ToolTalk service should aim to a string containing the contents of the property.

`int *len`
: Address of an integer that the ToolTalk service should set to the length of the value in bytes.

Returned Values

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_NUM
TT_ERR_PROPNAME
TT_ERR_SESSION
```

`unsigned char **value`
: Address of a character pointer that the ToolTalk service aimed at a string containing the contents of the property.

`int *len`
: Address of an integer that the ToolTalk service set to the length of the value in bytes.

tt_session_bprop_add

```
Tt_status    tt_session_bprop_add(const char *sessid,
             const char *propname,
             const unsigned char *value, int length)
```

Adds a new byte-string value to the end of the list of values for the named property of the session identified by `sessid`.

Arguments

`const char *sessid`
: Name of the session you have joined. Use the sessid value the ToolTalk service returns after you issue `tt_default_session()`.

`const char *propname`
: The name of the property to which you want to add values.

`const unsigned char *value`
: The value to add to the session property.

`int length`
: The size of the value in bytes.

Returned Values

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_PROPLEN
TT_ERR_PROPNAME
TT_ERR_SESSION
```

tt_session_bprop_set

```
Tt_status    tt_session_bprop_set(const char *sessid,
             const char *propname,
             const unsigned char *value, int length)
```

Replaces any current values stored under the named property of the session identified by sessid with the given byte-string value.

Arguments

`const char *sessid`
: Name of the session you have joined. Use the sessid value the ToolTalk service returns after you issue `tt_default_session()`.

`const char *propname`
: The name of the property whose value you want to replace.

`const unsigned char *value`
: The value to which the session property is set. If value is `NULL`, the property is removed entirely.

`int length`
: The size of the value in bytes.

Returned Values

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_PROPLEN
TT_ERR_PROPNAME
TT_ERR_SESSION
```

tt_session_join

```
Tt_status    tt_session_join(const char *sessid)
```

Joins the session named by the provided string and makes it the default session for your process.

Arguments

`const char *sessid`
: Name of the session you wish to join. Use the sessid value the ToolTalk service returns after you issue `tt_default_session()`, `tt_X_session()`, or `tt_initial_session()`.

Returned Values

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_PATH
```

Related Functions

`tt_default_session()`

tt_session_prop

```
char         *tt_session_prop(const char *sessid,
             const char *propname, int i)
```

Returns the *i*-th value (first value is number 0) of the specified session property.

Note: If this value has embedded nulls, you have no way to determine how long it is. Use `tt_session_bprop()` for values with embedded nulls.

Arguments

`const char *sessid`
: Name of the session you have joined. Use the sessid value the ToolTalk service returns after you issue `tt_default_session()`.

`const char *propname`
: The name of the property from which you want to retrieve a value. The name must be less than 64 characters.

`int i`
: The number of the item in the property name list for which you want to obtain the value. The list numbering begins with 0.

Returned Value

`char *`
: The value of the requested property. `NULL` is returned if there are *i* values or fewer. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_NUM
TT_ERR_PROPNAME
TT_ERR_SESSION
```

tt_session_prop_add

```
Tt_status    tt_session_prop_add(const char *sessid,
             const char *propname, const char *value)
```

Adds a new character-string value to the end of the list of values for the property of the specified session.

Arguments

`const char *sessid`
: Name of the session you have joined. Use the sessid value the ToolTalk service returns after you issue `tt_default_session()`.

`const char *propname`
: The name of the property to which you want to add a value. The name must be less than 64 characters.

`const char *value`
: The character string you want to add to the property name list.

Returned Values

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_PROPLEN
TT_ERR_PROPNAME
TT_ERR_SESSION
```

tt_session_prop_count

```
int          tt_session_prop_count(const char *sessid,
             const char *propname)
```

Returns the number of values stored under the named property of the session identified by sessid.

Arguments

`const char *sessid`
: Name of the session you have joined. Use the sessid value the ToolTalk service returns after you issue `tt_default_session()`.

`const char *propname`
: The name of the property you want to examine.

Returned Value

`int`
: The number of values in the specified property list. Use `tt_int_error()`, which returns `Tt_status`, to determine if the integer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_PROPNAME
TT_ERR_SESSION
```

tt_session_prop_set

```
Tt_status    tt_session_prop_set(const char *sessid,
             const char *propname, const char *value)
```

Replaces all current values stored under the named property of the session identified by sessid with the given character-string value.

Arguments

`const char *sessid`
: Name of the session you have joined. Use the sessid value the ToolTalk service returns after you issue `tt_default_session()`.

`const char *propname`
: The name of the property you want to examine.

`const char *value`
: The new value you want to insert. If you want to remove a value from the property list, specify the value as `NULL`.

Returned Values

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_PROPLEN
TT_ERR_PROPNAME
TT_ERR_SESSION
```

tt_session_propname

```
char         *tt_session_propname(const char *sessid,
             int n)
```

Return the *n*-th element of the list of currently-defined property names for the session identified by sessid.

Arguments

`const char *sessid`
: Name of the session you have joined. Use the sessid value the ToolTalk service returns after you issue `tt_default_session()`.

`int n`
: The number of the item in the property name list for which you want to obtain the name. The list numbering begins with 0.

Returned Value

char *

The name of the desired property from the session property list. `NULL` is returned if there are *n* properties or fewer. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_NUM
TT_ERR_SESSION
```

tt_session_propnames_count

int tt_session_propnames_count(
const char *sessid)

Return the number of currently-defined property names for the session.

Arguments

`const char *sessid`

Name of the session you have joined. Use the sessid value the ToolTalk service returns after you issue `tt_default_session()`.

Returned Value

`int`

The number of property names for the session. Use `tt_int_error()`, which returns `Tt_status`, to determine if the integer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_SESSION
```

tt_session_quit

```
Tt_status    tt_session_quit(const char *sessid)
```

Informs the ToolTalk service that your application is no longer interested in this ToolTalk session. The ToolTalk service will stop delivering messages scoped to this session.

Arguments

`const char *sessid`
: Name of the session you want to quit.

Returned Values

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_NOMP
TT_ERR_SESSION
TT_WRN_NOTFOUND
```

tt_spec_bprop

```
Tt_status    tt_spec_bprop(const char *objid,
             const char *propname, int i,
             unsigned char **value, int *length)
```

Retrieves the *i*-th value (first value is number 0) of this property.

Arguments

`const char *objid`
: Identifier of the object involved in this operation.

`const char *propname`
: Name of the property in which you are interested. The name must be less than 64 characters.

`int i`
: Item of the list in which you are interested. The list numbering begins with 0.

`unsigned char **value`
: Address of a character pointer that the ToolTalk service should aim to a string containing the contents of the spec's property. If there are *i* values or fewer, the pointer will be set to 0.

`int *len`
: Address of an integer that the ToolTalk service should set to the length of the value in bytes.

Returned Values

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_NUM
TT_ERR_OBJID
TT_ERR_PROPNAME
```

`unsigned char **value`
: Address of a character pointer that the ToolTalk service aimed at a string containing the contents of the property. If there are *i* values or fewer, the pointer will be set to 0.

`int *len`
: Address of an integer that the ToolTalk service set to the length of the value in bytes. If there are *i* values or fewer, the length will be 0.

tt_spec_bprop_add

```
Tt_status     tt_spec_bprop_add(const char *objid,
              const char *propname,
              const unsigned char *value, int length)
```

Adds a new byte-string to the end of the values list for this spec property.

Arguments

`const char *objid`
: Identifier of the object involved in this operation.

`const char *propname`
: Name of the property in which you are interested.

`const unsigned char *value`
: Byte string you want to add to the property value list.

`int length`
: Length of the value in bytes.

Returned Values

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_OBJID
TT_ERR_PROPLEN
TT_ERR_PROPNAME
```

tt_spec_bprop_set

```
Tt_status    tt_spec_bprop_set(const char *objid,
             const char *propname,
             const unsigned char *value, int length)
```

Replaces any current values stored under this spec property with a new byte-string.

Arguments

`const char *objid`
: Identifier of the object involved in this operation.

`const char *propname`
: Name of the property in which you are interested.

`const unsigned char *value`
Byte string you want to add to the property value list.

Note: If the value is `NULL`, the property is removed entirely.

`int length`
Length of the value in bytes.

Returned Values

Tt_statue
The status of the operation. Possible values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_OBJID
TT_ERR_PROPLEN
TT_ERR_PROPNAME
```

tt_spec_create

```
char          *tt_spec_create(const char *filepath)
```

Creates a spec (in memory) for an object. Use the objid that the ToolTalk service returns in future calls to manipulate the object.

Note: The object will not be a permanent ToolTalk item or visible to other processes until the creating process does a `tt_spec_write()`.

Arguments

`const char *filepath`
File name.

Returned Value

`char *`
Identifier for this object. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_OTYPE
TT_ERR_PATH
```

Related Functions

```
tt_spec_type_set()
tt_spec_write()
```

tt_spec_destroy

```
Tt_status    tt_spec_destroy(const char *objid)
```

Immediately destroys an object's spec.

Arguments

const char *objid
: Identifiifier of the object involved in this operationn.

Returned Values

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_OBJID
```

tt_spec_file

```
char         *tt_spec_file(const char *objid)
```

Retrieves the name of the file containing the object described by the spec.

Arguments

`const char *objid`
: Identifier of the object involved in this operation.

Returned Value

char *
: The file's absolute pathname. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_OBJID
```

tt_spec_move

```
char          *tt_spec_move(const char *objid,
              const char *newfilepath)
```

Notifies the ToolTalk service that this object has moved to a different file. The ToolTalk service returns a new objid for the object and a forwarding pointer is left from the old objid to the new one.

Note: If a new objid is not required (because the new and old files are in the same file system), `TT_WRN_SAME_OBJID` is returned.

For efficiency and reliability, your application should replace any references it has to the old objid with references to the new one.

Arguments

`const char *objid`
: Identifier of the object involved in this operation.

`const char *newfilepath`
: New file name.

Returned Value

`char *`

New unique identifier of the object involved in this operation. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_OBJID
TT_ERR_PATH
TT_WRN_SAME_OBJID
```

tt_spec_prop

```
char         *tt_spec_prop(const char *objid,
             const char *propname, int i)
```

Retrieves the i-th value (first value is number 0) of the property associated with this object spec.

Note: If this value has embedded nulls, you have no way to determine its length.

Arguments

`const char *objid`

Identifier of the object involved in this operation.

`const char *propname`

Name of the property in which you are interested.

`int i`

Item of the list in which you are interested. The list numbering begins with 0.

Returned Value

`char *`

Contents of the property value. A `NULL` value is returned if there are *i* values or less. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_NUM
TT_ERR_OBJID
TT_ERR_PROPNAME
```

tt_spec_prop_add

```
Tt_status    tt_spec_prop_add(const char *objid,
             const char *propname, const char *value)
```

Adds a new item to the end of the list of values associated with this spec property.

Arguments

`const char *objid`

Identifier of the object involved in this operation.

`const char *propname`

Property in which you are interested.

`const char *value`

New character-string to be added to the property value list.

Returned Values

`Tt_status`

The status of the operation. Possible values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
```

```
TT_ERR_OBJID
TT_ERR_PROPNAME
TT_ERR_PROPLEN
```

Related Functions

`tt_spec_prop_set()`

tt_spec_prop_count

```
int          tt_spec_prop_count(const char *objid,
             const char *propname)
```

Returns the number of values listed in this spec property.

Arguments

`const char *objid`
: Identifier of the object involved in this operation.

`const char *propname`
: Name of the property in which you are interested.

Returned Value

`int`
: Number of values listed in the spec property. Use `tt_int_error()`, which returns `Tt_status`, to determine if the integer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_OBJID
TT_ERR_PROPNAME
```

tt_spec_prop_set

```
Tt_status    tt_spec_prop_set(const char *objid,
             const char *propname, const char *value)
```

Replaces any values currently stored under this property of the object spec with a new value.

Arguments

`const char *objid`
: Identifier of the object involved in this operation.

`const char *propname`
: Name of the property in which you are interested.

`const char *value`
: Value you want to put in the property value list. If value is `NULL`, the property is removed entirely.

Returned Values

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_OBJID
TT_ERR_PROPNAME
TT_ERR_PROPLEN
```

Related Functions

`tt_spec_prop_add()`

tt_spec_propname

```
char         *tt_spec_propname(const char *objid, int n)
```

Returns the *n*-th element of the property name list for this object spec.

Arguments

`const char *objid`
: Identifier of the object involved in this operation.

`int n`

Item of the list in which you are interested. The list numbering begins with 0.

Returned Value

`char *`

Property name. `NULL` is returned if there are n properties or less.Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_NUM
TT_ERR_OBJID
```

tt_spec_propnames_count

```
int          tt_spec_propnames_count(const char *objid)
```

Returns the number of property names for this object.

Arguments

`const char *objid`

Identifier of the object involved in this operation.

Returned Value

`int`

Number of values listed in the spec property. Use `tt_int_error()`, which returns `Tt_status`, to determine if the integer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_OBJID
```

tt_spec_type

```
char         *tt_spec_type(const char *objid)
```

Returns the name (otid) of the object type.

Arguments

`const char *objid`
: Identifier of the object involved in this operation.

Returned Value

`char *`
: Type of this object. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_OBJID
```

tt_spec_type_set

```
Tt_status    tt_spec_type_set(const char *objid,
             const char *otid)
```

Assigns an object type (otype) value to the object spec. The type must be set before the spec is written for the first time, and cannot be changed thereafter.

Arguments

`const char *objid`
: Identifier of the object involved in this operation.

`const char *otid`
: Otype you want to assign to the spec.

Returned Values

`Tt_status`

The status of the operation. Possible values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_OBJID
TT_ERR_READONLY
```

Related Functions

```
tt_spec_create()
tt_spec_write()
```

tt_spec_write

```
Tt_status    tt_spec_write(const char *objid)
```

Writes the spec and any associated properties to the ToolTalk database. It is not necessary to do a write after a destroy.

Note: Several changes can be "batched" between write calls; for example, you might create an object spec, set some properties, and then write all the changes at once with one write call.

Arguments

`const char *objid`
: Identifier of the object involved in this operation.

Returned Values

`Tt_status`
: The status of the operation. Possible values are:

```
TT_OK
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_NOMP
TT_ERR_OBJID
```

Related Functions

```
tt_spec_create()
tt_spec_type_set()
```

tt_status_message

```
char          *tt_status_message(Tt_status ttrc)
```

Returns a pointer to a message describing the problem indicated by this Tt_status code.

Arguments

`Tt_status ttrc`
: `Tt_status` code you received during an operation.

Returned Value

`char *`
: Pointer to character string describing Tt_status code. Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

  `TT_OK`
  `TT_`*`xxx`*

tt_X_session

```
char          tt_X_session(const char *xdisplaystring)
```

Returns the session associated with the named X Window System display.

Arguments

`const char *xdisplaystring`
: Name of an X11 display server, e.g. somehost:0, :0, etc.

Returned Value

`char *`
: Identifier for the current ToolTalk session.

Use `tt_ptr_error()`, which returns `Tt_status`, to determine if the pointer is valid. Possible `Tt_status` values are:

```
TT_OK
TT_NOMP
```

Quick Reference to ToolTalk API

Table A-1 lists all functions in the ToolTalk API in alphabetical order. See Appendix B, "ToolTalk API Summary (Functional Grouping)," for a summary of the ToolTalk API by functional grouping, for example, all functions provided for creating a message.

Table A-2 lists the ToolTalk error-handling macros.

Return Type	ToolTalk Function
`Tt_status`	`tt_close(void)`
`char *`	`tt_default_file(void)`
`Tt_status`	`tt_default_file_set(const char *docid)`
`char *`	`tt_default_procid(void)`
`Tt_status`	`tt_default_procid_set(const char *procid)`
`char *`	`tt_default_ptype(void)`
`Tt_status`	`tt_default_ptype_set(const char *ptid)`
`char *`	`tt_default_session(void)`
`Tt_status`	`tt_default_session_set(const char *sessid)`
`int`	`tt_error_int(Tt_status ttrc)`
`void *`	`tt_error_pointer(Tt_status ttrc)`
`int`	`tt_fd(void)`
`Tt_status`	`tt_file_copy(const char *oldfilepath, const char *newfilepath)`
`Tt_status`	`tt_file_destroy(const char *filepath)`

Table A-1 ToolTalk API Summary (Alphabetical)

Return Type	ToolTalk Function
Tt_status	tt_file_join(const char *filepath)
Tt_status	tt_file_move(const char *oldfilepath, const char *newfilepath)
Tt_status	tt_file_objects_query(const char *filepath, Tt_filter_function filter, void *context, void *accumulator)
Tt_status	tt_file_quit(const char *filepath)
void	tt_free(caddr_t p)
char *	tt_initial_session(void)
Tt_status	tt_int_error(int return_val)
caddr_t	tt_malloc(size_t s)
int	tt_mark(void)
Tt_address	tt_message_address(Tt_message m)
Tt_status	tt_message_address_set(Tt_message m, Tt_address p)
Tt_status	tt_message_arg_add(Tt_message m, Tt_mode n, const char *vtype, const char *value
Tt_status	tt_message_arg_bval(Tt_message m, int n, unsigned char **value, int *len)
Tt_status	tt_message_arg_bval_set(Tt_message m, int n, const unsigned char *value, int len)
Tt_status	tt_message_arg_ival(Tt_message m, int n, int *value)
Tt_status	tt_message_arg_ival_set(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 A-1 ToolTalk API Summary (Alphabetical)

Return Type	ToolTalk Function
char *	tt_message_arg_val(Tt_message m, int n)
Tt_status	tt_message_arg_val_set(Tt_message m, int n, const char *value)
int	tt_message_args_count(Tt_message m)
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_callback_add(Tt_message m, Tt_message_callback f)
Tt_class	tt_message_class(Tt_message m)
Tt_status	tt_message_class_set(Tt_message m, Tt_class c)
Tt_message	tt_message_create(void)
Tt_message	tt_message_create_super(Tt_message m)
Tt_status	tt_message_destroy(Tt_message m)
Tt_disposition	tt_message_disposition(Tt_message m)
Tt_status	tt_message_disposition_set(Tt_message m, Tt_disposition r)
Tt_status	tt_message_fail(Tt_message m)
char *	tt_message_file(Tt_message m)
Tt_status	tt_message_file_set(Tt_message m, const char *file)
gid_t	tt_message_gid(Tt_message m)
char *	tt_message_handler(Tt_message m)
char *	tt_message_handler_ptype(Tt_message m)
Tt_status	tt_message_handler_ptype_set(Tt_message m, const char *ptid)

Table A-1 ToolTalk API Summary (Alphabetical)

Return Type	ToolTalk Function
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)
char *	tt_message_object(Tt_message m)
Tt_status	tt_message_object_set(Tt_message m, const char *objid)
char *	tt_message_op(Tt_message m)
Tt_status	tt_message_op_set(Tt_message m, const char *opname)
int	tt_message_opnum(Tt_message m)
char *	tt_message_otype(Tt_message m)
Tt_status	tt_message_otype_set(Tt_message m, const char *otype)
Tt_pattern	tt_message_pattern(Tt_message m)
Tt_message	tt_message_receive(void)
Tt_status	tt_message_reject(Tt_message m)
Tt_status	tt_message_reply(Tt_message m)
Tt_scope	tt_message_scope(Tt_message m)
Tt_status	tt_message_scope_set(Tt_message m, Tt_scope s)
Tt_status	tt_message_send(Tt_message m)
char *	tt_message_sender(Tt_message m)
char *	tt_message_sender_ptype(Tt_message m)
Tt_status	tt_message_sender_ptype_set(Tt_message m, const char *ptid)
char *	tt_message_session(Tt_message m)

Table A-1 ToolTalk API Summary (Alphabetical)

Return Type	ToolTalk Function
Tt_status	tt_message_session_set(Tt_message m, const char *sessid)
Tt_state	tt_message_state(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)
uid_t	tt_message_uid(Tt_message m)
void *	tt_message_user(Tt_message m, int key)
Tt_status	tt_message_user_set(Tt_message m, int key, void *v)
int	tt_objid_equal(const char *objid1, const char *objid2)
char *	tt_objid_objkey(const char *objid)
Tt_message	tt_onotice_create(const char *objid, const char *op)
char *	tt_open(void)
Tt_message	tt_orequest_create(const char *objid, const char *op)
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)

Table A-1 ToolTalk API Summary (Alphabetical)

Return Type	ToolTalk Function
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)
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)
Tt_status	tt_pattern_address_add(Tt_pattern p, Tt_address d)
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_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)

Table A-1 ToolTalk API Summary (Alphabetical)

Return Type	ToolTalk Function
Tt_pattern	tt_pattern_create(void)
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_iarg_add(Tt_pattern m, Tt_mode n, const char *vtype, int value)
Tt_status	tt_pattern_object_add(Tt_pattern p, const char *objid)
Tt_status	tt_pattern_op_add(Tt_pattern p, const char *opname)
Tt_status	tt_pattern_otype_add(Tt_pattern p, const char *otype)
Tt_status	tt_pattern_register(Tt_pattern p)
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)
Tt_status	tt_pattern_unregister(Tt_pattern p)
void *	tt_pattern_user(Tt_pattern p, int key)
Tt_status	tt_pattern_user_set(Tt_pattern p, int key, void *v)

Table A-1 ToolTalk API Summary (Alphabetical)

Return Type	ToolTalk Function
Tt_message	tt_pnotice_create(Tt_scope scope, const char *op)
Tt_status	tt_pointer_error(void *pointer)
Tt_message	tt_prequest_create(Tt_scope scope, const char *op)
Tt_status	tt_ptype_declare(const char *ptid)
void	tt_release(int mark)
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)
Tt_status	tt_session_join(const char *sessid)
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)
char *	tt_session_propname(const char *sessid, int n)
int	tt_session_propnames_count(const char *sessid)
Tt_status	tt_session_quit(const char *sessid)

Table A-1 ToolTalk API Summary (Alphabetical)

Return Type	ToolTalk Function
Tt_status	tt_spec_bprop(const char *objid, const char *propname, int i,unsigned char **value, int *length)
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)
char *	tt_spec_create(const char *filepath)
Tt_status	tt_spec_destroy(const char *objid)
char *	tt_spec_file(const char *objid)
char *	tt_spec_move(const char *objid, const char *newfilepath)
char *	tt_spec_prop(const char *objid, const char *propname, int i)
Tt_status	tt_spec_prop_add(const char *objid, const char *propname, const char *value)
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)
char *	tt_spec_propname(const char *objid, int n)
int	tt_spec_propnames_count(const char *objid)
char *	tt_spec_type(const char *objid)
Tt_status	tt_spec_type_set(const char *objid, const char *otid)
Tt_status	tt_spec_write(const char *objid)

Table A-1 ToolTalk API Summary (Alphabetical)

Return Type	ToolTalk Function
`char *`	`tt_status_message(Tt_status ttrc)`
`char *`	`tt_X_session(const car *xdisplay)`

Table A-1 ToolTalk API Summary (Alphabetical)

Return Type	ToolTalk Functions
`int`	`tt_is_err(Tt_status s)`
`Tt_status`	`tt_ptr_error(pointer)`

Table A-2 ToolTalk Macros

Appendix B

ToolTalk API Summary (Functional Grouping)

This appendix lists all functions in the ToolTalk API in the groups required to perform certain operations. See Appendix A, "Quick Reference to ToolTalk API," for a summary of the ToolTalk API by alphabetical listing.

The API functions are grouped under these headings:

- "Initialization"
- "Message Patterns"
- "Session"
- "Files"
- "Messages"
- "Objects"
- "ToolTalk Storage Management"
- "ToolTalk Error Status"
- "Exiting"
- "ToolTalk Error-Handling Macros"

Initialization

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_porcid_set(const char *porcid)
char *	tt_default_ptype(viod)
Tt_status	tt_defualt_ptype_set(const char*ptid)
Tt_status	tt_ptype_declare(const char *ptid)
int	tt_fd(void)

Table B-1 Initializing and Registering with the ToolTalk Service

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)

Table B-2 Creating, Filling In, Registering, and Destrouing Message Patterns

Return Type	ToolTalk Function
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)
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_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)
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)

Table B-2 Creating, Filling In, Registering, and Destrouing Message Patterns

Return Type	ToolTalk Function
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_pattern_destroy(Tt_pattern p)

Table B-2 Creating, Filling In, Registering, and Destrouing Message Patterns

Session

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)
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)

Table B-3 Expressing Interest in Sessions

Return Type	ToolTalk Function
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)

Table B-4 Managing Session Information

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)
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)

Table B-5 Expressing Interest in Files

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 B-6 Creating Messages

Return Type	ToolTalk Function
Tt_status	tt_message_address_set(Tt_message m, Tt_address p)
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_val_set(Tt_message m, int n, const char *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_callback_add(Tt_message m, Tt_message_callback f)

Table B-7 Filling In Messages and Replies

Return Type	ToolTalk Function
Tt_status	tt_message_class_set(Tt_message m, Tt_class c)
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_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_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)

Table B-7 Filling In Messages and Replies

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)
char *	tt_message_arg_val(Tt_message m, int n)
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_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)

Table B-8 Examining Messages

Return Type	ToolTalk Function
`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_messae m)`
`uid_t`	`tt_message_user(Tt_message m, int key)`
`void *`	`tt_message_send(Tt_message m)`
`Tt_status`	`tt_message_send(Tt_message m)`
`Tt_status`	`tt_message_destroy(Tt_message m)`

Table B-8 Examining Messages

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

Table B-9 Sending 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)

Table B-10 Receiving, Replying to, Rejecting, and Destroying Messages

Return Type	ToolTalk Function
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)

Table B-10 Receiving, Replying to, Rejecting, and Destroying Messages

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 B-11 Creating, Moving, and Destroying Objects

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)
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 B-12 Using ToolTalk Storage

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)
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)

Table B-13 Examining Object Type Information

ToolTalk Storage Management

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)

Table B-14 Managing ToolTalk Storage

ToolTalk Error Status

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 B-15 Retrieving ToolTalk Error Information

Return Type	ToolTalk Function
int	tt_error_int(Tt_status ttrc)
void *	tt_error_pointer(Tt_status ttrc)

Table B-16 Encoding Error Values

Exiting

Return Type	ToolTalk Function
Tt_status	tt_close(void)

Table B-17 Leaving the ToolTalk Session

ToolTalk Error-Handling Macros

Return Type	ToolTalk Function
int	tt_is_err(Tt_status s)
Tt_status	tt_ptr_error(pointer)

Table B-18 ToolTalk Error Handling Errors

Initialization Error Messages

The following ToolTalk error messages can occur either when the ToolTalk service, or an application that uses the ToolTalk service, is attempting to start up.

`/bin/sh: application_name: not found`

The start string as installed in the types database does not correspond to an executable file in `$PATH`.

To start the application:

First, ask your user to start the application as he or she would start it without the ToolTalk service.

After the application has started, ask your user to retry the operation that should have started the application with the ToolTalk service.

`Cannot open display`

ttsession could not contact the server named with the -d *display* option or the $DISPLAY variable.

To open the display:

Verify that the named display is running. (See the `X(1) man` page.)

Verify that the host on which you are running `ttsession` has permission to connect to it. (See the `X(1)`, `xhost(1)`, and `xauth(1) man` pages.

Appendix D

ToolTalk Error Messages

The ToolTalk error and warning message identifiers are allocated as follows.

Table D-1 ToolTalk Error and Warning Message Identifiers

Message or Group	Range
TT_OK	0
TT_WRN_*	1- 511
APP_WRN_*	512 - 1023
TT_WRN_LAST	1024
TT_ERR_*	1025 - 1535
APP_ERR_*	1536 - 2046
TT_ERR_LAST	2047

TT_WRN_NOTFOUND

Message ID	TTERR-0
Catalog String	`TT_OK` Request successful.
Meaning	Your call was completed successfully.

TT_WRN_NOTFOUND

Message ID	TTERR-1
Catalog String	`TT_WRN_NOTFOUND` The object was not removed because it was not found.
Meaning	When the ToolTalk service could not find the specified object in the ToolTalk database. The destroy operation did not succeed.

TT_WRN_STALE_OID

Message ID	TTERR-2
Catalog String	`TT_WRN_STALE_OID` The object attribute in the message has been replaced with a newer one. Update the place from which the object id was obtained.
Meaning	When the ToolTalk service looked up the specified object in the ToolTalk database, it found a forwarding pointer to the object.
Remedy	The ToolTalk service automatically puts the new objid in the message. Use `tt_message_object()` to retrieve the new objid. Update any internal application references to the new objid.

TT_WRN_STOPPED

Message ID	TTERR-3
Catalog String	`TT_WRN_STOPPED` The query was halted by the filter procedure.
Meaning	The query operation being performed was halted by `Tt_filter_function`.

TT_WRN_SAME_OBJID

Message ID	TTERR-4
Catalog String	`TT_WRN_SAME_OBJID` The moved object retains the same objid.
Meaning	The object you moved stayed within the same file system. The ToolTalk service will retain the same objid and update the location.

TT_WRN_START_MESSAGE

Message ID	TTERR-5
Catalog String	`TT_WRN_START_MESSAGE` This message caused this process to be started. This message should be replied to even if it is a notice.
Meaning	When the ToolTalk service starts your application to deliver a message, you must reply, even if the message is a notice.
Remedy	Use `tt_message_reply()` to reply to the message you received after the process was started by the ToolTalk service.

TT_WRN_APPFIRST

Message ID	TTERR-512
Catalog String	`TT_WRN_APPFIRST` This code should be unused.
Meaning	This message id marks the beginning of the messages allocated for ToolTalk application warnings.

TT_WRN_LAST

Message ID	TTERR-1024
Catalog String	`TT_WRN_LAST` This code should be unused.
Meaning	This message id marks the last of the messages allocated for ToolTalk warnings.

TT_ERR_CLASS

Message ID	TTERR-1025
Catalog String	`TT_ERR_CLASS` The `Tt_class` value passed is invalid.
Meaning	The ToolTalk service does not recognize the class value you specified.
Remedy	The `Tt_class` values are `TT_NOTICE` and `TT_REQUEST`. Retry the call with one of these values.

TT_ERR_DBAVAIL

Message ID	TTERR-1026
Catalog String	`TT_ERR_DBAVAIL` A required database is not available. The condition may be temporary, trying again later may work.
Meaning	The ToolTalk service could not access the ToolTalk database needed for this operation.
Remedy	Try the operation again later. Check if the file server or workstation that contains the database is available.

TT_ERR_DBEXIST

Message ID	TTERR-1027
Catalog String	`TT_ERR_DBEXIST` A required database does not exist. The database must be created before this action will work.
Meaning	The ToolTalk service did not find the specified ToolTalk database in the expected place.
Remedy	Install the `rpc.ttdbserverd` program on the machine that stores the file or object involved in this operation. See *Tooltalk Setup and Administration Guide* for instructions.

TT_ERR_FILE

Message ID	TTERR-1028
Catalog String	`TT_ERR_FILE` File object could not be found.
Meaning	The file specified does not exist or is not accessible.
Remedy	Check your file path name and retry the operation. Check if the machine where the file is stored is accessible.

TT_ERR_MODE

Message ID	TTERR-1031
Catalog String	`TT_ERR_MODE` The Tt_mode value is not valid.
Meaning	The ToolTalk service does not recognize the specified mode value.
Remedy	The `Tt_mode` values are `TT_IN`, `TT_OUT`, and `TT_INOUT`. Retry the call with one of these values.

TT_ERR_ACCESS

Message ID	TTERR-1032
Catalog String	`TT_ERR_ACCESS` An attempt was made to access a ToolTalk object in a way forbidden by the protection system.
Meaning	The user does not have the necessary access to the object and the process, and therefore, cannot perform the operation. For example, the user may not have permission to destroy an object spec.
Remedy	The user needs to gain proper access to the object before the application can perform the operation.

TT_ERR_NOMP

Message ID	TTERR-1033
Catalog String	`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.
Meaning	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.
Remedy	Verify that `ttsession` is installed on the machine in use.

TT_ERR_NOTHANDLER

Message ID	TTERR-1034
Catalog String	`TT_ERR_NOTHANDLER` Only the handler of the message can do this.
Meaning	Only the handler of a message can perform this operation. Your application is not the handler for this message.

TT_ERR_NUM

Message ID	TTERR-1035
Catalog String	`TT_ERR_NUM` The integer value passed is not valid.
Meaning	An invalid integer value that was very out-of-range was passed to the ToolTalk service. Simple out-of-range conditions, such as requesting the third value of a property that has only two values, return a null value.
Remedy	Check the integer you specified.

TT_ERR_OBJID

Message ID	TTERR-1036
Catalog String	`TT_ERR_OBJID` The object id passed does not refer to any existing object spec.
Meaning	The ToolTalk service found the objid in the ToolTalk database but it does not reference an existing object.
Remedy	Clean up the ToolTalk database with the `ttdbck` utility.

TT_ERR_OP

Message ID	TTERR-1037
Catalog String	`TT_ERR_OP` The operation name passed is not syntactically valid.
Meaning	The specified operation name is null or contains non-alphanumeric characters.
Remedy	Remove any non-alphanumeric characters and retry the operation.

TT_ERR_OTYPE

Message ID	TTERR-1038
Catalog String	`TT_ERR_OTYPE` The object type passed is not the name of an installed object type.
Meaning	The ToolTalk service could not locate the specified otype.
Remedy	Check the type of the object with `tt_spec_type()`. If the application was recently installed and the ToolTalk service has not re-read the types database, locate the process id for the `ttsession`, and force the re-read with the USR2 signal. **`% ps -elf | grep ttsession`** **`% kill -USR2 <ttsession pid>`**

TT_ERR_ADDRESS

Message ID	TTERR-1039
Catalog String	`TT_ERR_ADDRESS` The `Tt_address` value passed is not valid.
Meaning	The ToolTalk service does not recognize the address value you specified.
Remedy	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

Message ID	TTERR-1040
Catalog String	`TT_ERR_PATH` One of the directories in the file path passed does not exist or cannot be read.
Meaning	The ToolTalk service was not able to read a directory in the specified file path name.
Remedy	Check the pathname to ensure that the current user has access to the specified directories. Check the machine where the file resides to make sure it is accessible.

TT_ERROR_OTYPE

Message ID	TTERR-1041
Catalog String	`TT_ERR_POINTER` The opaque pointer (handle) passed does not indicate an object of the proper type.
Meaning	The pointer you 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.
Remedy	Check the arguments for the ToolTalk function to find what arguments the function expects. Retry the operation with a pointer for a valid object.

TT_ERR_PROCID

Message ID	TTERR-1042
Catalog String	`TT_ERR_PROCID` The process id passed is not valid.
Meaning	The process identifier you specified is out of date or invalid.
Remedy	Retrieve the default procid with `tt_default_procid()`.

TT_ERR_PROPLEN

Message ID	TTERR-1043
Catalog String	`TT_ERR_PROPLEN` The property value passed is too long.
Meaning	The ToolTalk service accepts property values of up to 64 characters.
Remedy	Shorten the property value to less than 64 characters.

TT_ERR_PROPNAME

Message ID	TTERR-1044
Catalog String	`TT_ERR_PROPNAME` The property name passed is syntactically invalid.
Meaning	The property name is too long, contains non-alphanumeric characters, or is null.
Remedy	Check the property name, modify if necessary, and retry the operation.

TT_ERR_PTYPE

Message ID	TTERR-1045
Catalog String	`TT_ERR_PTYPE` The process type passed is not the name of an installed process type.
Meaning	The ToolTalk service could not locate the specified ptype.
Remedy	If the application was recently installed and the ToolTalk service has not re-read the types database, locate the process id for the `ttsession`, and force the re-read with the USR2 signal. **`% ps -elf | grep ttsession`** **`% kill -USR2 <ttsession pid>`**

TT_ERR_DISPOSITION

Message ID	TTERR-1046
Catalog String	`TT_ERR_DISPOSITION` The `Tt_disposition` value passed is not valid.
Meaning	The disposition you passed is not recognized by the ToolTalk service.
Remedy	The `Tt_disposition` values are `TT_DISCARD`, `TT_QUEUE`, and `TT_START`. Retry the call with one of these values.

TT_ERR_SCOPE

Message ID	TTERR-1047
Catalog String	`TT_ERR_SCOPE` The `Tt_scope` value passed is not valid.
Meaning	The scope you passed is not recognized by the ToolTalk service.
Remedy	The `Tt_scope` values are `TT_SESSION` and `TT_FILE`. Retry the call with one of these values.

TT_ERR_SESSION

Message ID	TTERR-1048
Catalog String	`TT_ERR_SESSION` The session id passed is not the name of an active session.
Meaning	You specified an out of date or invalid ToolTalk session.
Remedy	Use `tt_default_session()` to obtain the sessid of the current default session or use `tt_initial_session()` to obtain the sessid of the initial session your application was started in.

TT_ERR_VTYPE

Message ID	TTERR-1049
Catalog String	`TT_ERR_VTYPE` The value type name passed is not valid.
Meaning	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`.
Remedy	Change the type of the value to either `int` or `string` and retry the operation.

TT_ERR_NO_VALUE

Message ID	TTERR-1050
Catalog String	`TT_ERR_NO_VALUE` No property value with the given name and number exists.
Meaning	The ToolTalk service could not locate the specified property value you specified in the ToolTalk database.
Remedy	Retrieve the current list of properties to find the property you want.

TT_ERR_INTERNAL

Message ID	TTERR-1051
Catalog String	`TT_ERR_INTERNAL` Internal error (bug)
Meaning	The ToolTalk service has suffered an internal error.
Remedy	Restart all applications that are using the ToolTalk service. Report the error to SGI Customer Support.

TT_ERR_READONLY

Message ID	TTERR-1052
Catalog String	`TT_ERR_READONLY` The attribute cannot be changed.
Meaning	The attribute your application is trying to change is not owned or writable by the current user.

TT_ERR_NO_MATCH

Message ID	TTERR-1053
Catalog String	`TT_ERR_NO_MATCH` No handler could be found for this message, and the disposition was not queue or start.
Meaning	The message your application sent could not be delivered. No applications that are running have registered interest in this type of message.
Remedy	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.

TT_ERR_UNIMP

Message ID	TTERR-1054
Catalog String	`TT_ERR_UNIMP` Function not implemented.
Meaning	The ToolTalk function called is not implemented.

TT_ERR_OVERFLOW

Message ID	TTERR-1055
Catalog String	`TT_ERR_OVERFLOW` Too many active messages (try again later).
Meaning	The ToolTalk service has received the maximum amount of active messages (2000) it can handle properly.
Remedy	Retrieve any messages that the ToolTalk service may be queueing for your application. Send your message again later. `ttsession` can also be started 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

Message ID	TTERR-1056
Catalog String	`TT_ERR_PTYPE_START` Attempt to launch instance of ptype failed.
Meaning	The ToolTalk service could not start the type of process specified.
Remedy	Check to see that the application that the ptype represents is properly installed.

TT_ERR_APPFIRST

Message ID	TTERR-1536
Catalog String	`TT_ERR_APPFIRST` This code should be unused.
Meaning	This message id marks the beginning of the messages allocated for ToolTalk application errors.

TT_ERR_LAST

Message ID	TTERR-2047
Catalog String	`TT_ERR_LAST` This code should be unused.
Meaning	This message id marks the last of the messages allocated for ToolTalk errors.

TT_STATUS_LAST

Message ID	TTERR-2048
Catalog String	`TT_STATUS_LAST` This code should be unused.
Meaning	This message id marks the last of the messages allocated for ToolTalk status.

Index

D

E

F

H

R

S

T

U

V

W

X