Integrating Applications with the SPARCworks 3.0 Toolset

May 1994

SunPro
A Sun Microsystems, Inc. Business

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part Number: 801-7208-01
Revision 4.0

Contents

Preface

Intended Audience

This manual is intended for software developers who wish to integrate additional software development tools into SPARCworks™ 3.0. SPARCworks 3.0 supports ToolTalk as an interapplication, message-passing mechanism by which to communicate with other software tools. The SPARCworks suite of tools contains a set of protocols which a developer can use to integrate an application with SPARCworks. These protocols define messages sent and received by each of the SPARCworks tools.

Note: The messages described in this white paper are also applicable in the ProWorks environment.

Organization

Chapter 1, "SPARCworks Tool Integration," provides an overview of the SPARCworks tools and describes how to integrate your tools into the SPARCworks Toolset.

Chapter 2, "The ToolTalk Service," provides an overview of the ToolTalk service and its general use in tool integration. This chapter describes how to instrument your code to open up a connection with ToolTalk. The chapter also describes how to register message patterns that you would like to send and receive.

Chapter 3, “Integrating an Application with SPARCworks,” describes all the ToolTalk messages available from the SPARCworks 3.0 tools. You can use these messages to integrate your applications with SPARCworks.

Appendix A, “ToolTalk Message Index,” provides an alphabetical listing of all ToolTalk messages available from SPARCworks 3.0 with cross references to the text pages on which the messages are described.

Additional References

For more information on SPARCworks, consult the following publications:

- *SPARCworks 3.0 Manual Set*, Part Number 825-2189-02
- *SPARCworks Technical White Paper: SPARCworks Toolset*, Part Number 800-6943-10
- *Product Guide: Power Tools for Professional Developers*

For more information on ToolTalk for Solaris 2. 2 users, consult the following references:

- *ToolTalk 1.1 User's Guide*, Part Number 801-3667-10
- *ToolTalk 1.1 Reference Manual*, Part Number 801-3666-10
- SunSoft Press. *The ToolTalk Service: An Inter-operability Solution.*

For more information on ToolTalk for Solaris 2. 3 users, consult the following references:

- *ToolTalk 1.1.1 User's Guide*, Part Number 801-5275-1
- *ToolTalk 1.1.1 Reference Manual*, Part Number 801-5275-10
- SunSoft Press. *The ToolTalk Service: An Inter-operability Solution.*

SPARCworks Tool Integration

1

Introduction to SPARCworks

SPARCworks™ 3.0 is an open, integrated suite of programming tools for use with SPARCompiler™ language systems. The SPARCworks programming environment speeds and simplifies the tasks software developers do most often — edit and merge source code, compile programs, debug programs, and tune program performance.

The following SPARCompiler language systems can be used with SPARCworks tools:

- SPARCompiler C
- SPARCompiler C++
- SPARCompiler FORTRAN
- SPARCompiler Pascal

A separate version of SPARCworks is available to support SPARCompiler Ada.

SPARCworks can be extended to support multithreaded application development with SPARCworks/iMPact. SPARCworks can also be extended to support code management with SPARCworks/TeamWare.

The ToolTalk Service

SPARCworks™ 3.0 supports ToolTalk - the COSE standard for inter-tool communication - as an interapplication, message-passing mechanism by which to communicate with other software tools. The SPARCworks suite of tools contains a set of protocols which you can use to integrate your tools with SPARCworks. These protocols define messages sent and received by each of the SPARCworks tools. SPARCworks itself is integrated using ToolTalk.

More about SPARCworks

The SPARCworks programming environment consists of the following six tools:

- **SPARCworks Manager —** for managing and coordinating other SPARCworks programming tools
- **SourceBrowser —** for statically analyzing source code and program structure
- **Debugger —** for dynamically analyzing programs, observing behavior while the program runs, and collecting information for performance profiling
- **Analyzer —** for tuning program performance, including memory allocation
- **FileMerge —** for merging source files and coordinating source code changes
- **MakeTool —** for building programs and browsing makefiles

SPARCworks Manager

SPARCworks Manager provides an easy-to-use, unified way to start SPARCworks tools and control the programming environment. A tool that is started from the SPARCworks Manager is thereafter associated with that instance of SPARCworks Manager, and can be controlled as part of a management session.

In addition to managing the other five standard SPARCworks tools, SPARCworks Manager can also control custom applications written by users or third-party developers. When integrated with SPARCworks and executed under SPARCworks Manager, the standard SPARCworks tools and custom applications combine to provide you with the following advantages:

- **Drag and drop execution —** You can drag an on-screen DeskSet object (a file icon, for example) with the mouse pointer and drop it on a SPARCworks tool, which recognizes whether or not the object can be loaded or otherwise acted upon.

- **Session control —** You can group development activities into convenient management sessions. Multiple sessions can proceed simultaneously, each controlled by its own SPARCworks Manager.
- **Workspace organization —** SPARCworks Manager helps you make the best use of on-screen workspace. SPARCworks tools associated with a management session can be closed to a single icon.
- **Central control —** The SPARCworks Manager can set environment variables and the working directory for all SPARCworks tools used in a session.

SourceBrowser

SourceBrowser is the SPARCworks tool for static program analysis. It offers multiple language support, including programs that mix software modules written in different languages. It provides a Text Editor for source, enabling you to edit source using Text Editor commands similar to those in other OpenWindows applications, such as MailTool. SourceBrowser handles large applications, maintaining a high level of efficiency regardless of the amount of code you are browsing.

You can use SourceBrowser as a GUI to `grep` and display a source file containing matches. SourceBrowser enables you to find symbols and strings in source and to match wildcards and regular expressions. You can also perform semantic queries.

Using SourceBrowser, you can browse information about programs that do not compile completely. You can also produce function call graphs. For C++, you can also produce class graphs that represent the inheritance hierarchy and browse class definitions.

Debugger

The SPARCworks Debugger dynamically analyzes a program, letting you observe its behavior while it is running. Debugger gives you complete control of program execution and simultaneously collects performance data for later use with SPARCworks Analyzer. From within Debugger you can identify a problem, edit source code, rebuild the program, and then continue dynamic analysis.

In addition to the mouse-selectable functions of the OpenWindows interface (which you can customize to suit your needs), Debugger provides a pane in which you can type commands directly to the command line.

Advanced Debugger features include integrated runtime error checking to improve quality, and fix and continue to dramatically improve developer productivity.

SPARCworks/iMPact is an optional product to enhance Debugger support for developing and debugging multithreaded applications.

Analyzer

After having successfully compiled a program and eliminated its major bugs, you can evaluate its performance using the SPARCworks Analyzer. The Analyser measures and displays an application's performance profile, suggesting ways to improve performance.

The Analyzer eliminates the need to compile and link an application with special data collection instrumentation — any program that has been compiled for debugging can be analyzed. You can collect a variety of performance data types, and you can control the data collection process while an application is running. By letting you focus on the areas where performance problems occur, the Analyzer easily tests your hypotheses about a program's behavior.

As a further help in rebuilding your program with improved performance, the Analyzer identifies improved ordering for loading functions into the program's address space. It creates a mapfile and you can then rebuild the program by passing the new ordering to the Solaris linker and producing an executable with reduced working set size.

FileMerge

SPARCworks FileMerge provides a convenient way to merge two text files or directories of files. Most FileMerge functionality is duplicated in the Solaris `diff`(1) utility, but the FileMerge window interface helps merge files more easily and quickly than any command-line based tool.

In addition to loading two files to be merged, a user can specify a third file, called the ancestor of the two files. The two files to be merged are called descendants of the ancestor. When an ancestor file has been specified, FileMerge marks lines in the descendants that are different from the ancestor and produces a merged file based on all three files.

MakeTool

MakeTool is an OPEN LOOK interface to `make`(1), the Solaris utility that oversees program compilation and ensures that programs are compiled from the newest sources. In addition, MakeTool contains a browser that helps you interpret makefiles by expanding macros and rules.

Advantages of SPARCworks Tools

SPARCworks tools, with their high performance and full feature set, enhance the already rich Solaris software development environment. Together, SPARCworks tools provide the following advantages:

- The consistent look and feel provided by the standard OPEN LOOK interface shortens learning time for new users.
- Visual and functional compatibility with DeskSet™ tools such as File Manager and Text Edit give SPARCworks tools added value in the Sun development environment.
- SPARCworks Manager provides a uniform way to integrate new tools into the SPARCworks toolset and coordinate them with the standard SPARCworks tools.
- For simultaneous static and dynamic program analysis, SourceBrowser and Debugger form a powerful combination.
- Advanced Debugger features such as runtime error checking, and fix and continue combine to improve quality and reduce turnaround time.
- The concise graphical presentation of data provided by SourceBrowser and Analyzer quickly clarifies program structure and uncovers performance bottlenecks.
- The process of building programs and maintaining makefiles is assisted by MakeTool, a graphical interface for the popular `make` utility.
- Differences between source files can be found easily and a merged version of two source files can be created quickly with FileMerge.

Integrating with SPARCworks

SPARCworks Manager session control and SPARCworks tool interaction is accomplished through use of the Sun ToolTalk service. The ToolTalk service is a network-spanning, interapplication communication service that allows applications to communicate with other autonomous applications. Special SPARCworks protocols have been designed from the larger ToolTalk service to provide the means for SPARCworks Manager, SourceBrowser, FileMerge, and Debugger to communicate with and transfer data to your tool.

Types of Integration

In the same way that SPARCworks tools are integrated together, third party developers can integrate their applications with SPARCworks. The three types of integration available are:

- Visual Integration
- Control Integration
- Data Integration

Your application may include one or more types of integration. None of the integration types is a prerequisite for the other.

Visual Integration

At the level of visual integration, all tools are accessible from the SPARCworks Manager tool palette. All tools are invoked in a similar way, by double clicking on the icon. Cut/copy/paste is supported across tools. One advantage of having a common feel is the ease with which developers new to SPARCworks can learn the tools and come up to speed. Tools on the palette are in a common ToolTalk session, with the advantage that tools respond to a minimum subset of ToolTalk messages, such as open and close.

Control Integration

At the level of control integration, all tools communicate using ToolTalk. Control integration means that one tool can invoke operation of another tool. For example, from the SPARCworks CallGrapher selecting a function and pressing the source button causes the source for the selected function to be displayed in the SourceBrowser. The tool control integration is accomplished using ToolTalk messages. Third party tools can use these same messages to integrate with the SPARCworks toolset.

Another example is the SPARCworks Debugger, which displays information from the underlying `dbx` debugger engine based on ToolTalk messages.

Data Integration

Data integration involves taking advantage of data either provided by the compilers or produced by the SPARCworks tools. Such data comes in the form of the following:

- **stabs Format**: The SPARCompilers generate program data in stabs format which is utilized by the Debugger. stabs provides comprehensive information about the structure and data of your program. If your tool is source-code oriented, then stabs can help you with data integration. The *SPARCworks 2.0 Debugger Interface* manual describes the stabs format and is available through SunPro's Partners Integration Program.
- **IDF Format**: The SourceBrowser is capable of generating a very simple-to-parse data file of information found in the SourceBrowser database. The format of this file is called Intermediate Data Format (IDF). Though the information is not as comprehensive as the stabs format, it is easier to parse and understand. You would use this method if your source code data needs are simple and you would like to integrate with the SourceBrowser. The IDF file format is documented in the manual *Browsing Source Code*. This manual is part of the *SPARCworks 3.0 Manual Set.*
- **Performance Data**. The Performance Analyzer is able to generate an ASCII dump of the performance data collected. The data, generated by running the `er_export(1)` utility, can be easily parsed and, perhaps, displayed in a multitude of ways not provided by the Performance Analyzer. The format of the `er_export` output is documented in the manual *Performance Tuning an Application*. This manual is part of the *SPARCworks 3.0 Manual Set.*

Strategies for Integration

As a prospective integrator, think first about the types of integration that will be most useful to your customers. Next, map out the steps necessary to accomplish that kind of integration.

Here's one suggestion for how to start integrating with SPARCworks:

- Understand the SPARCworks toolset
- Decide what process you want to automate
- Learn ToolTalk basics
- Examine the ToolTalk messages for specific SPARCworks tools
- Start with visual integration (if applicable)

Understand the SPARCworks toolset

This document explains some of the basic features of the SPARCworks tools and their integration possibilities. For more detailed information on the SPARCworks tools, consult the SPARCworks documentation set.

Decide what process you want to automate

Would you like the user to be able to start your tools and SPARCworks and manage their entire development session from one place? Is there particular information about code that you would like to have, such as symbols and types, function names, or performance data? Would the dynamic state of the Debugger be useful to you? Would you like to execute a command on behalf of the user, and allow the command's process to be tracked by the user? There are many ways in which the SPARCworks toolset can add value to your environment.

Learn ToolTalk basics

Chapter 2, "The ToolTalk Service," describes the basics of ToolTalk and provides sample code fragments for setting up a ToolTalk session.

Examine the ToolTalk messages for SPARCworks tools

Examine the ToolTalk messages for the specific SPARCworks tools with which you would like to integrate. Chapter 3, "Integrating an Application with SPARCworks," is organized by each SPARCworks tool. The messages listed are those available for you to use in your integration.

Start with visual integration

Start with visual integration by integrating your tool with the SPARCworks Manager to acquaint yourself with the integration process. Once that is accomplished, then control and/or data integration will be achieved more easily.

The ToolTalk Service[1]

2

This section describes how the *ToolTalk* service, a network-spanning, interapplication communication service, allows your application to communicate with other autonomous applications. Tutorial-style instructions for modifying your application to communicate via ToolTalk messages are given in the latter half of this section.

The ToolTalk service provides multicast messaging; that is, an application sends a message that is delivered by the ToolTalk service to multiple receivers. Multicast messaging, with the concept of one-to-many communications, falls between broadcast messaging (one-to-all) and point-to-point messaging (one-to-one). The ToolTalk service also provides point-to-point messaging between applications.

The ToolTalk service supports two types of messaging, process-oriented and object-oriented messaging. Process-oriented messages are addressed to other processes; object-oriented messages are addressed to objects managed by processes.

This section introduces you to multicast, process-oriented messaging and how to modify your application to send and receive these messages. For more information on object-oriented messaging and the ToolTalk service in general, refer to the *ToolTalk 1.1.1 User's Guide.*

.

1. This section is adapted from the SunSoft *DeskSet Integration Guide*.

The ToolTalk Service Overview

The ToolTalk service is used by independent applications to communicate with each other without having direct knowledge of each other. Applications communicate by creating and sending ToolTalk messages. The ToolTalk service receives these messages, determines the recipients, and then delivers the messages to the appropriate applications, as illustrated in Figure 2-1.

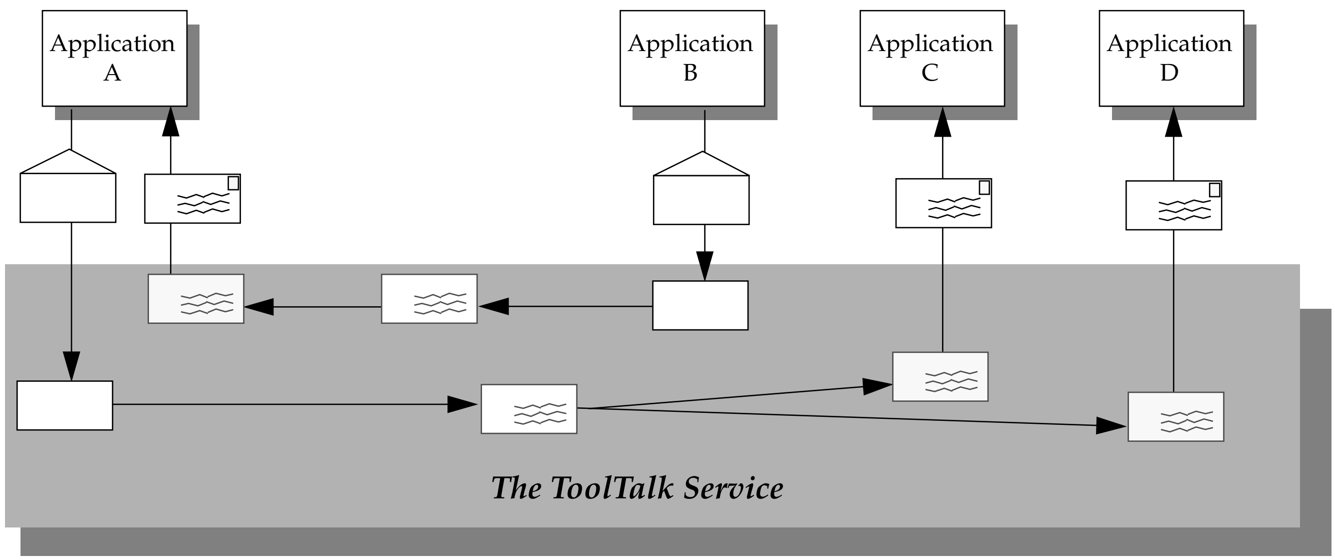

Figure 2-1 Applications Using The ToolTalk Service To Communicate

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

How Applications Use the ToolTalk Service

To communicate, applications create, send, and receive ToolTalk messages. Sending applications create, fill in, and send a message; the ToolTalk service determines the recipient and delivers the message to the receiving application.

Receiving applications retrieve messages and, after examining the information in the message, either perform an operation and reply with the results or discard the message.

An important ToolTalk feature is that the sending application needs to know little about the receiving application. This is because applications that want to receive messages register their interest in specific types of messages with the ToolTalk service.

Registering Interest In ToolTalk Messages

Sending applications need to know little about the receiving application because applications that want to receive messages explicitly state what these messages should look like. This information is registered with ToolTalk in the form of *message patterns*. These message patterns usually match the message protocols that applications have agreed to use. Applications can add more patterns for individual use.

Message patterns are created much like a message is created; the same type of information is used in both. 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.

When the ToolTalk service receives a message from a sender, it compares the information in the message to the patterns that are registered. Once matches have been found, the ToolTalk service delivers copies of the message to all recipients.

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

Applications provide message patterns to the ToolTalk service at installation time and/or while the application is running. The latter half of this section provides instructions on how to modify your program.

Sending ToolTalk Messages

To send a ToolTalk message, an application obtains an empty message, fills in the message attributes, and sends the message. ToolTalk messages are simple structures containing fields for address, subject, and delivery information. To send a ToolTalk message, an application need only provide this information:

- Should the recipient respond to the message; is this a notice or a request?
- What operation should be performed/has been performed? Are arguments needed to perform the operation?
- What interest does the recipient share with the sender; is the recipient running in a specific user session or interested in a specific file?

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

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. The process retrieves the message and examines its contents. The message may contain a notice that some operation has been performed. In this case, the process discards the message after reading the information. 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 can discard the original message.

Process-Oriented Messaging

Process-oriented messaging provides a communication path between applications to deliver information or request that an operation be performed by the process receiving the message. To use process-oriented messaging, you need to be familiar with these ToolTalk concepts:

- Processes
- Sessions

Processes

One execution of an application, tool, or program that uses the ToolTalk service is called a *process* in this section. A process is a Solaris process that has initialized and registered with the ToolTalk service.

Procid

When a process opens communication with the ToolTalk service, it receives a *process identifier* (*procid*). Applications that send messages to a specific process must know the receiving application's procid. To find out this information, the sender can look at a previous message sent by the target application. The ToolTalk service automatically fills in a message's sender attribute with the sender's procid.

Ptype

You can instruct the ToolTalk service to consider your application as a potential message receiver when no process is running the application. This is done by providing message patterns and instructions for starting the application in a *process type* (*ptype*). The ptype file is compiled with the ToolTalk type compiler, *tt_type_comp*, at application installation time to make the information available to the ToolTalk service.

When an application includes a ptype, part of the registration and initialization activities register the ptype with the ToolTalk service. Registering a ptype will automatically register the message patterns listed in it.

In this section, the instructions for modifying your program explain how to register message patterns while the application is running. See the *ToolTalk 1.1.1 User's Guide* for more information on ptypes.

Sessions

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

When a process opens communication with the ToolTalk service, the process that actually provides the procid is ttsession. A side effect of opening ToolTalk communication is that the session in which the action takes place becomes the default session for the process.

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

ToolTalk Messages

ToolTalk messages contain attributes that store message information and provide delivery information to the ToolTalk service. ToolTalk uses this delivery information to route the messages to the appropriate receivers.

Message Attributes

ToolTalk messages are simple structures containing attributes for address, subject (*operation* and *arguments*), and delivery information (*class and scope*.) Each message consists of attributes such as those shown in Table 2-1.

Table 2-1 Basic Message Attributes

Attribute	Value
Address	procedure, process, object, object type
Class	request or notice
Operation	action to be performed, or event that has occurred
Arguments	input values for operation, output results
Scope	session

These are fundamental attributes. More message attributes are available for use by an application. See the *ToolTalk 1.1.1 User's Guide* for a complete listing of message attributes.

Addressing

Messages addressed to other applications can be addressed to a particular process (*process* address) or to any process that has registered a pattern that matches your message (*procedure* address). When addressing a message to a process, you need to know the process identifier (procid) of the other application. Applications receive a procid when they open communication with ToolTalk. The procid is unique within the user's session in which the application was started.

However, it is unusual for one process to know another's procid; more often a sender doesn't care which process performs an operation (request message) or learns of an event (notice message).

Notices and Requests

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.

Scope

Applications using the ToolTalk service to communicate usually have something in common – the applications are running in the same session or they're interested in the same file or data. Applications register this interest by joining sessions with the ToolTalk service. This session information is used by the ToolTalk service in conjunction with the message patterns to determine what applications should receive a message.

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

How the ToolTalk Service Routes Messages

A notice takes a one-way trip, as shown in Figure 2-2. The sender creates a message, fills in attribute values, and sends it. The ToolTalk service matches message and pattern attribute values, and gives a copy of the message to one handler and to all matching observers.

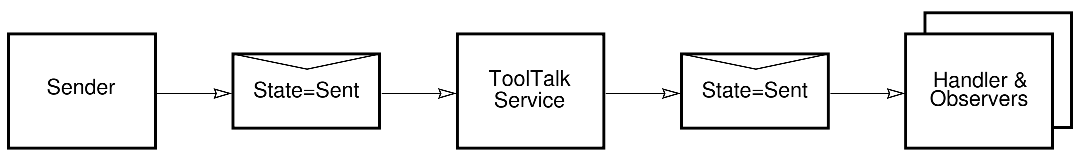

Figure 2-2 Notice Routing

A request, as shown in Figure 2-3, takes a round-trip from sender to handler and back; copies of the message take one-way side trips to observers. The ToolTalk service delivers a request to, at most, one handler. The handler adds results to the message and sends it back. Other processes can observe a request before it is handled, after, or both; observers absorb a request without sending it back.

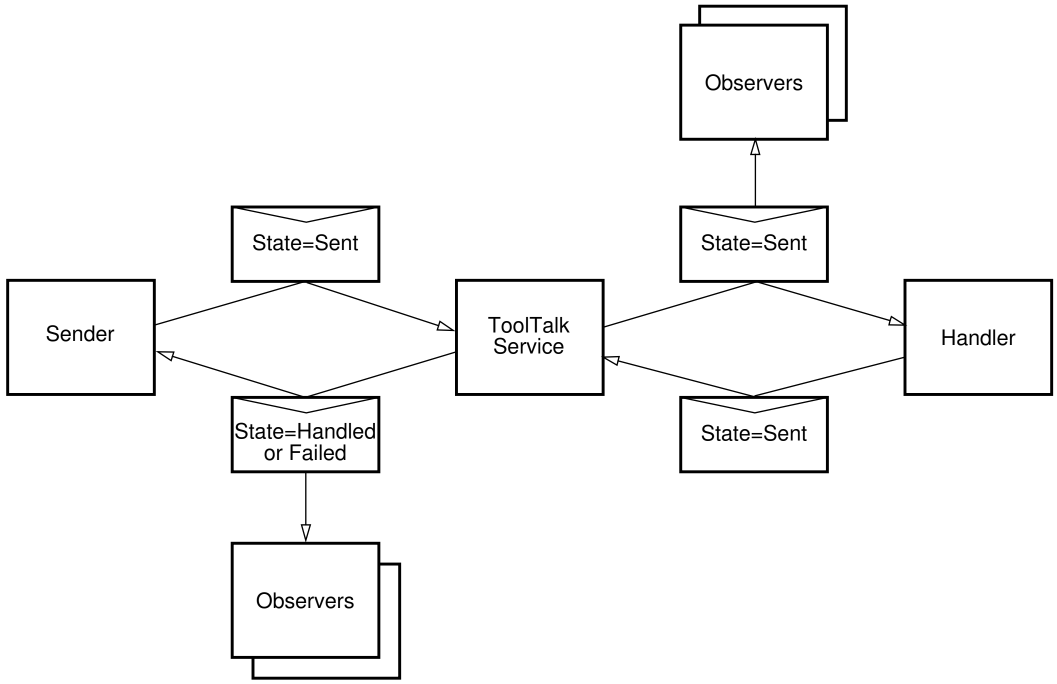

Figure 2-3 Request Routing

Modifying Your Application to Use the ToolTalk Service

The steps you take in modifying your application to use the ToolTalk service fall into these groups:

- Including the API header files
- Preparing for interaction with the ToolTalk service
- Using message patterns
- Registering interest in a session or file
- Sending messages
- Receiving messages
- Stopping interaction with the ToolTalk service

To illustrate how an application has been modified to use the ToolTalk service, a sample program, `ttsample1`, is used in the following sections. This program is available in the following OpenWindows directory:

```
$OPENWINHOME/share/src/tooltalk/
```

To run the sample program, make the program in the directory.

```
% make ttsample1
```

Include the ToolTalk API File

Include the `tt_c.h` file in your program. It's located in this directory:

```
%OPENWINHOME/share/include/desktop/
```

Here are the included files in `ttsample1`.

```
/*
 * ttsample1 -- dynamic pattern, procedural notification
 */

#include <stdio.h>
#include <sys/param.h>
#include <sys/types.h>
#include <xview/xview.h>
#include <xview/panel.h>
#include <strings.h>

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

Preparing for Interaction with the ToolTalk Service

To prepare for interacting with the ToolTalk Service, you will

- Open communication with the ToolTalk Service
- Obtain a file descriptor that will notify you when messages have arrived
- Setup to receive messages

The functions used in these operations are listed in Table 2-2.

Table 2-2 ToolTalk Initialization Functions

Function	Return Value Type	Description
`tt_open`	char*	Opens communication, sets default session to current user session, returns procid
`tt_fd`	int	Obtains file descriptor

Open Communication and Obtain File Descriptor

To initialize your process with the ToolTalk service and prepare it to receive messages, you must obtain a process identifier (procid) and a file descriptor (fd). The file descriptor informs your process when messages are delivered. `tt_open()` returns the procid for your process and sets this procid as the default procid.

`tt_fd()` returns a file descriptor. When a message arrives for the default procid in the default session, the file descriptor becomes active.

Here's sample code to initialize and register with the ToolTalk service.

```
/*
 * 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();
```

Setting Up to Receive Messages

When a message has arrived for your application, the file descriptor becomes active. How you are alerted that the file descriptor is active will vary depending on how your application is structured.

XView™ Programs

A program that uses the XView notifier, through `xv_main_loop()` or `notify_start()`, can have a callback function invoked when the file descriptor becomes active. Invoke `notify_set_input_func()` with the handle for the message object as a parameter.

Here's how `ttsample1`, an XView program, is set up to receive messages.

```
ttfd = tt_fd();

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

notify_set_input_func(base_frame, receive_tt_message, ttfd);
```

X Window System Xt (Intrinsics) Programs

A Xt-based program uses `XtAddInput` to watch for arriving messages.

TNT Programs

A TNT-based program uses `wire_AddFileHandler` to watch for an active file descriptor.

Other Xlib Programs

Programs structured around a `poll` use the file descriptor returned by `tt_fd()`. When the file descriptor becomes active, the `poll` will exit.

Using Message Patterns To Register Interest In Messages

After setting up to receive messages, you need to tell the ToolTalk service what types of messages you want to receive. To create and register message patterns that the ToolTalk service will use when determining message recipients, you

- Obtain a message pattern handle
- Set and add pattern attributes

- Register the message pattern
- Unregister the message pattern when you no longer want to receive messages that match this pattern

The functions used in these operations are listed in Table 2-3.

Table 2-3 Message Pattern Functions

Function	Return Value Type	Description
tt_pattern_create	Tt_pattern	Obtains a handle to a message pattern
tt_pattern_<*attribute*>_set	Tt_status	Sets the value for this pattern attribute
tt_pattern_<*attribute*>_add	Tt_status	Adds values to this pattern attribute
tt_pattern_register	Tt_status	Registers pattern with the ToolTalk service
tt_pattern_unregister	Tt_status	Unregisters pattern from the ToolTalk service
tt_pattern_destroy	Tt_status	Destroys the message pattern

Obtain a Message Handle

To get a "handle" or "opaque pointer" to a new pattern object, use tt_pattern_create(). Use this handle on succeeding calls to reference the pattern.

Set and Add Pattern Attributes

To fill in pattern information, use the tt_pattern_<*attribute*>_add() or tt_pattern_<*attribute*>_set() calls for each attribute of the pattern. See Table 2-4 for pattern attributes relevant to process-oriented messages.

Table 2-4 Message Pattern Attributes for _add

Attribute	Values	Description
arg	char[]	Add arguments
barg	unsigned char[]	Add boolean arguments
iarg	int	Add integer arguments
address	TT_PROCEDURE, TT_OBJECT, TT_HANDLER, TT_OTYPE	Add address
callback	function_name	Add message callback

class	TT_NOTICE, TT_REQUEST	Add class
disposition	TT_DISCARD, TT_QUEUE, TT_START, TT_QUEUE+TT_START	Add disposition for receiving tool
file	char *file	Add filename
object	char *objid	Add object
op	char *opname	Add operation
opnum	int opnum	Add operation number
otype	char *object_type	Add object type
scope	TT_SESSION, TT_FILE, TT_BOTH, TT_FILE_IN_SESSION	Add scope
sender	char *procid	Add sending process identifier
sender_ptype	char process_type	Add sending process type
session	char *sessid	Add session identifier
state	TT_CREATED, TT_SENT, TT_HANDLED, TT_FAILED, TT_QUEUED, TT_REJECTED, TT_STARTED	Add state

Table 2-5 Message Pattern Attributes for _set

Attribute	**Values**	**Description**
category	TT_OBSERVE,T_HANDLE	Set category
user	void* v	Set arbitrary user data (one word length)

Note – You can supply multiple values for each attribute in the pattern except for category; the pattern attribute matches a message attribute if any of the values in the pattern match the value in the message.

The following pattern attributes must always be supplied:

- Category
- Scope

Registering a Message Pattern

When the pattern is complete, register it with `tt_pattern_register()`, and join the sessions or files you specified.

Here's sample code to create and register 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);
```

Unregistering Message Patterns

To stop receiving messages that match a message pattern, use `tt_pattern_unregister()` to unregister the pattern or `tt_pattern_destroy()` to unregister and then destroy the pattern object.

The ToolTalk service will automatically destroy message pattern objects when `tt_close()` is called.

Registering Interest in a Session or File

Note – All SPARCworks messaging is session-scoped, not file-scoped. Information on joining a file is included in this section for other ToolTalk users who may be aware of this capability.

After registering message patterns, join a session and/or file to update the session and file attributes in your message pattern. If you specified a scope of `TT_SESSION` in your message pattern, you need to join the session to notify the ToolTalk service to add your sessid to the sessid attribute of the message pattern. This automatic update of message patterns by joining sessions or files saves the effort of registering a new pattern for each session or file your application is interested in.

To join or quit a session or file, use one of the functions listed in Table 2-6.

Table 2-6 Registering Interest Functions

Function	Return Value Type	Description
`tt_session_join`	Tt_status	Registers interest in the indicated user session
`tt_session_quit`	Tt_status	Unregisters interest in the indicated user session
`tt_file_join`	Tt_status	Registers interest in the indicated file
`tt_file_quit`	Tt_status	Unregisters interest in the indicated file

Session

Use `tt_session_join()` to register interest in messages that name a specific session monitored by the ToolTalk service. When you join, supply the sessid of the session you want to join. Joining a session automatically updates all of your session-scoped message patterns with the specified sessid.

Here's sample code for joining the default user session in which this program was started.

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

tt_session_join(tt_default_session());
```

When you no longer want to participate in the default session, inform the ToolTalk service with `tt_session_quit()`. The sessid will be removed from your session-scoped message patterns.

File

As mentioned previously, all SPARCworks messaging is session-scoped, not file-scoped. There is no reason to join a file at this time. This section is included for comprehensiveness, only.

To register your interest in messages about a particular file, call `tt_file_join()`. Joining a file automatically updates all of your filescoped message patterns with the name of the file. The file name is added to a list; it does not replace existing file names.

When you're no longer interested in receiving messages that refer to the file, call `tt_file_quit()`. The file name will be removed from your file-scoped message patterns.

Sending Messages

To create, fill in, and send a message, use the functions listed in Table 2-7.

Table 2-7 Sending Messages Functions

Function	**Return Value Type**	**Description**
`tt_pnotice_create`	Tt_message	Creates a process-oriented notice and returns a handle to the message
`tt_prequest_create`	Tt_message	Creates a process-oriented request and returns a handle to the message
`tt_message_`<*attribute*>`_set`	Tt_status	Sets the value for a message attribute
`tt_message_`<*attribute*>`_add`	Tt_status	Adds a value for a message attribute
`tt_message_send`	Tt_status	Sends a message
`tt_message_destroy`	Tt_status	Destroys a message that is no longer useful

Create and Fill In Message

To get a "handle" or "opaque pointer" to a new message object for a process notice or request, use `tt_pnotice_create()` or `tt_prequest_create()`. Use this handle on succeeding calls to refer to the message.

To fill in message information, use the `tt_message_`<*attribute*>`_set()` calls for the following attributes.

- 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 fill in the default session and/or file.

- Op

  Fill in the operation that describes the notice or request you are making. To determine the operation name, consult the `message` protocol definition. The SPARCworks 3.0 message protocol definitions are described in Chapter 2, "Integrating an Application into SPARCworks Manager."

In addition to these required attributes, you can fill in other attributes, such as operation arguments.

Send the Message

To send a message, use `tt_message_send()`. If you no longer need the message (for example, you are not expecting a reply) destroy the message and free memory with `tt_message_destroy()`.

Here's the code for creating and sending a pnotice from the `ttsample1` program.

```
/*
 * Create and send a ToolTalk notice message
 * ttsample1_value(in int new_value)
 */

msg_out = tt_pnotice_create(TT_SESSION, "ttsample1_value");
tt_message_arg_add(msg_out, TT_IN, "integer", NULL);
tt_message_arg_ival_set(msg_out, 0, (int)xv_get(slider,
PANEL_VALUE));
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);
}
```

Receiving Messages

To retrieve a message, use the functions listed in Table 2-8.

Table 2-8 Receiving Messages Functions

Function	Return Value Type	Description
`tt_message_receive`	Tt_message	Retrieves the currently available message
`tt_mark`	mark	Marks position on ToolTalk stack to be used when releasing information when finished
`tt_release`	<none>	Uses mark from which to free all storage relating to the message
`tt_free`	void	Frees storage from the API stack
`tt_message_`<*attribute*>	(depends on attribute)	Obtains the value for a message attribute
`tt_message_destroy`	Tt_status	Destroys a message that is no longer useful

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

When you use functions that return information such as the `tt_message_`<*attribute*>`()` calls, the ToolTalk service creates a copy of the information (the original copy is in the message structure in this case) and stores the copy in an allocation stack in the API library. The ToolTalk service actually returns a pointer to the copy to you. If you don't need the information returned by the function, use `tt_free()` to free the storage in the API stack.

The ToolTalk API also provides a more general mechanism, the `tt_mark()` and `tt_release()` functions, to mark and free information returned by a series of functions. 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 interesting after the routine has ended.

Note – The API allocation stack should not be confused with your program's run-time stack. The ToolTalk service returns information from ToolTalk functions by providing pointers to the information in the API stack. The API stack will not discard information until you tell it to.

In the following example, `ttsample1` calls `tt_mark()` at the beginning of the routine that examines the information in a message. The ToolTalk service returns a mark, an integer that represents a location on the API stack. 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.

Use the `tt_message_`<*attribute*>`()` calls to examine the attributes of the message to determine the action you should take.

You can recognize replies to messages you sent by comparing the handles. This is why it is important not to destroy a request after you send it. When you have finished examining a message you received, be sure to destroy the message using `tt_message_destroy()` to free memory allocated for the message.

Here's the code from `ttsample1` for receiving 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;


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

        mark = tt_mark();

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

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

Stopping Interaction with the ToolTalk Service

Use `tt_close()` when you want to stop interacting with the ToolTalk service and other ToolTalk session participants. `tt_close()` closes the current default procid. Here's how the `ttsample1` program uses this call.

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

exit(0);
}
```

Integrating an Application with SPARCworks 3

This section describes the ToolTalk service messages that are used to connect individual tools that are part of the SPARCworks 3.0 programming environment.

This chapter includes the following sections:

Messages Description Syntax

The messages are shown using a simple and easily parsable syntax, as follows:

message ::= ‘(‘`Tt_message` *attribute** [*args*] ‘)’

attribute ::= ‘(‘*field_name value* ‘)’

args ::= ‘(‘`Tt_args` *arg+* ‘)’

arg ::= ‘(‘*direction* [*type*] *value* ‘)’

direction ::= `TT_IN` | `TT_OUT` | `TT_INOUT`

type ::= `quoted string`

Values in uppercase are ToolTalk enumeration constants, names beginning with “`Tt_`” identify ToolTalk field names, and those in lowercase are variable names. The second level of nesting is because arguments do not have separate field names and position in the sequence is significant. Also because of the distinction between *direction* and *field_name*.

For purposes of describing messages in the document (but not for actually generating messages) three additional notations are defined:

- Optional arguments are enclosed in square [] brackets, and are preceded by an integer argument that indicates whether or not an optional argument is present in the message. Where there are multiple arguments separated by a pipe within the square brackets, the user must choose between one of the arguments.
- Variables are indicated in italics.
- Variable length lists are indicated by bracketing possibly separated groups of arguments with {}* for zero or more repetitions and {}+ for one or more repetitions.

Assumed Message Items

Unless other values are given for these attributes, the following two default values are used in the messages described below:

(`Tt_address TT_PROCEDURE`) (`Tt_scope TT_SESSION`)

Messages with the TT_PROCEDURE default will be broadcast to any recipient which registers the pattern.

SPARCworks Session Protocol

Starting SPARCworks Tools

SPARCworks 3.0 tools are started and controlled in two ways, either:

- Directly by the user, or
- By the SPARCworks Manager

When the SPARCworks Manager is used to start a tool, the SPARCworks Manager and the tool must exchange information about each other in order to subsequently communicate properly when using the ToolTalk service. SPARCworks Manager must know the tool's process ID. The tool must know:

- The SPARCworks Manager's process ID
- Its own unique ID key
- The SPARCworks session name

When SPARCworks Manager itself is started, it sets the value of the environment variable `SW_TT_STARTUP_PROCID` to the value of its own process ID.

1. When SPARCworks Manager launches the tool, it passes it an unique ID string (key) as the argument to the `-swtm` option.
2. The tool receives the key and sends a `PEI_STARTED` ToolTalk message back to the originating SPARCworks Manager. The message contains the key and the tool's own process ID. It determines the SPARCworks Manager's process ID by reading the `SW_TT_STARTUP_PROCID` shell environment variable.
3. The SPARCworks Manager replies to the message and supplies the tool with the SPARCworks session name.

A tool that has been started by the SPARCworks Manager may, in turn, start additional tools; the automatic pass-down of the environment variable will connect them with a SPARCworks Manager.

Note – The `-swtm` argument is not required for proper operation, but is provided so that SPARCworks Manager can track the tool it starts.

Alternatively, a tool may acquire more control over tools it starts by changing the environment variable and receiving the `PEI_STARTED` message itself.

Start-up Message

When the SPARCworks Manager first starts a tool, the tool announces its existence by sending the PEI_STARTED message.

```
(Tt_message
   (Tt_class TT_REQUEST) (Tt_op "PEI_STARTED")
   (Tt_address TT_HANDLER)(Tt_handler swmgr_procid)
   (Tt_args (TT_IN key)
      (TT_OUT sw_session)))
```

swmgr_procid	SPARCworks Manager's process ID as received from the `SW_TT_STARTUP_PROCID` environment variable. See the `getenv(3C)` man page for information on retrieving environment variable settings.
key	Key string as passed to tool as argument of `-swtm` option
sw_session	Session name returned by SPARCworks Manager

SPARCworks Manager Integration

Visual integration and session control can be accomplished through the SPARCworks Manager. For example, any icon can be easily included in the SPARCworks Manager palette and session-controlled through ToolTalk. The amount of additional code needed is minimal. The advantage to this visual integration is that, since it is the simplest among the SPARCworks integrations, it provides a starting ground for integrating with SPARCworks. The SPARCworks Manager palette provides a "one-stop shop" from which to start and manage all the applications on the users' desktop.

The SPARCworks Manager messages are commands which are sent to one or more tools running in the SPARCworks session. They instruct the tool to raise, unmap, open, and close its windows as well as to quit. They include `expose`, `hide`, `PEI_OPEN`, `PEI_CLOSE`, `quit` and `departed`.

Exposing a Tool

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "expose")
   (Tt_args (TT_IN "string" sw_session)))
```

sw_session	Session name (a tool only acts on this message if this name matches the value received as the *sw_session* argument returned in the `PEI_STARTED` message)

Hiding a Tool

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "hide")
   (Tt_args (TT_IN "string" sw_session)))
```

sw_session	Session name (a tool only acts on this message if this name matches the value received as the *sw_session* argument returned in the `PEI_STARTED` message)

Opening a Window

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "PEI_OPEN")
   (Tt_args (TT_IN "string" sw_session)))
```

sw_session	Session name (a tool only acts on this message if this name matches the value received as the *sw_session* argument returned in the `PEI_STARTED` message)

Closing a Window

```
(Tt_message
      (Tt_class TT_NOTICE) (Tt_op "PEI_CLOSE")
      (Tt_args (TT_IN "string" sw_session)))
```

sw_session	Session name (a tool only acts on this message if this name matches the value received as the *sw_session* argument returned in the `PEI_STARTED` message)

Tool Quits

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "quit")
   (Tt_args (TT_IN "string" sw_session)))
```

sw_session	Session name (a tool only acts on this message if this name matches the value received as the *sw_session* argument returned in the `PEI_STARTED` message)

Tool Departed

When a tool is about to exit, it will inform the SPARCworks Manager using:

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "departed")
   (Tt_args (TT_IN "string" sw_session)))
```

sw_session	Session name (the value received as the *sw_session* argument returned in the `PEI_STARTED` message)

SourceBrowser Integration

Both control integration and data integration can be utilized in the SourceBrowser. For example, if information about source code is displayed in your tool, graphically or otherwise, it is possible to tell the SourceBrowser to query for a particular symbol in a specified file by issuing the `SPRO_Visit_Object` message. This is useful for loading source code which might be interesting to the user as it provides the ability to modify the code within the SourceBrowser.

Control integration in the opposite direction can be accomplished through the `IDE_browse` message. A symbol selected in the SourceBrowser window can be passed along with the name of its source file to your tool. This helps the user gain access to your data from the SourceBrowser.

Information from the SourceBrowser database can be generated into an easily parsable file by using the `IDE_gen_design` message. This method allows your application to have data integration by gaining access to calling hierarchy and parameter information from the SourceBrowser database. The format of this file is documented in the manual *Browsing Source Code*.

In this release of the protocol, users who have IDE's C Development Environment (CDE) can take advantage of the integration between CDE and SPARCworks. The IDE Integration messages include messages for tool coordination, data exchange, and display synchronization.

Control and Data Integration

The control and data integration messages include `SPRO_SBENG_START`, `SPRO_SBENG_QUITS`, and `SPRO_SBENG_CURRENT_ELEMENT`.

SourceBrowser Engine Starts

When the SourceBrowser engine starts, it sends:

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "SPRO_SBENG_START"))
```

SourceBrowser Engine Quits

When the SourceBrowser engine quits it sends:

```
(Tt_message
        (Tt_class TT_NOTICE) (Tt_op "SPRO_SBENG_QUITS"))
```

SourceBrowser Source Display Changes

When the current match of the SourceBrowser source display changes, the SourceBrowser sends:

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op
"SPRO_SBENG_CURRENT_ELEMENT")
   (Tt_args (TT_IN "string" context_uid)
      (TT_IN "string" filename)
      (TT_IN "int" lineno)
      (TT_IN "int" center)))
```

context_uid	Ignored, reserved for future use
filename	The file that is being displayed
lineno	The line number that is being displayed
center	Must be zero. Reserved for future use

IDE Integration

This section describes the messages passed between CDE and the SourceBrowser. `IDE_quit` is the single tool coordination message. Three data exchange messages include `IDE_gen_design`, `IDE_sync`, and `IDF_generated`. Two display synchronization messages include `SPRO_Visit_Object` and `IDE_browse`.

CDE and SourceBrowser are not automatically started in order to handle a request. In particular, the SourceBrowser must already be started before CDE can send it an IDE_gen_design, IDE_sync, or SPRO_Visit_Object message. Furthermore, CDE's cnav tool must be already started before the SourceBrowser can send it an IDE_browse message.

Tool Coordination

CDE Exits

The SourceBrowser does not check to determine if `cnav` was started successfully. `cnav` must be started through the CDE user interface.

```
(Tt_message
      (Tt_class TT_NOTICE) (Tt_op "IDE_quit"))
```

This notice is sent by CDE when it exits. The SourceBrowser receives, but then ignores this message.

Errors

In the protocol defined in the next two sections (for data exchange and for display synchronization), all error handling will be dealt with using the ToolTalk error status functionality. That is, in replying to one of the requests defined below, an error will be registered by filling out the ToolTalk error status code and adding an error status string.

ToolTalk itself will also make use of the error status code for errors and warnings in the control of ToolTalk. An example of this is when a message is not handled by an application.

Each application (CDE and SourceBrowser) can then do anything with the error message and code they wish. Presumably, they will print out the error message to the user (e.g., in the footer of a window).

The status codes will be defined by an enum (see below) that will be used in the error status field of the ToolTalk message. There is an associated error string with each error code, but that can vary at the discretion of the application that "owns" the specific error code.

The status code enum follows. This can be put in a header file used by each application. Any changes to this error code enum would, of course, have to be coordinated between the applications (CDE and SourceBrowser).

```
typedef enum sb_ide_status {
   SB_IDE_OK = 0,      /* TT_OK */
   SB_IDE_ERR = 1792, /* TT_ERR_APPFIRST + 256 */
   SB_IDE_ERR_NODIR,   /* Directory not valid */
   SB_IDE_ERR_NODB,    /* No database found */
```

```
    SB_IDE_ERR_NOWRITE,/* Directory not writable */
    SB_IDE_ERR_NODUMP, /* Failed to generate IDF dump */
    SB_IDE_ERR_NOFILE, /* Source file not found */
    SB_IDE_ERR_INVOBJ, /* Invalid object type */
    SB_IDE_ERR_NOSYMF, /* Name not found in file */
    SB_IDE_ERR_NOSYMD, /* Name not found in database */
    SB_IDE_ERR_INTERN  /* Internal error */
} Sb_ide_status;
```

The following macro can be used to allow each application to have a simple check for an IDE/SourceBrowser error:

```
#define tt_is_sb_ide_err(p) (SB_IDE_ERR < (p))
```

For example,

```
if (tt_is_sb_ide_err(code)) {...}.
```

Of course, the following ToolTalk macro will still work for ToolTalk errors as well as CDE/SourceBrowser errors:

```
tt_is_error(code)
```

The sample code below illustrates how these codes can be used. The first sample is a callback that will send a reply to a request. The reply will add an error code and string.

```
Tt_callback_action
got_a_request_callback(Tt_message m, Tt_pattern p)
{
    int ans;

    /* attempt to process message with, say, a message arg */
    ans = do_it(tt_message_arg_val(m, 0);

    /* if couldn’t, setup status error */
    if (ans) {
        tt_message_status_set(m, SB_IDE_ERR_NODB);
        tt_message_status_string_set(m, “No database found”);
    }   else {
        /* else set some out args of the message... */
        set_out_args(m);
    }

    /* send reply */
    tt_message_reply(m);

    /* done */
    tt_message_destroy(m);
    return TT_CALLBACK_PROCESSED;
}
```

The second example is a callback that will receive a reply with an error code and string.

```
Tt_callback_action
my_request_was_answered(Tt_message m, Tt_pattern p)
{
    char *string = NULL;/* error string */
    Sb_ide_status status;/* status code... status is just int */
        /* could have explicit casts everywhere, though, e.g.
         *     status = (Sb_ide_status)tt_message_status(m) and
         *     tt_is_error((Tt_status)status) etc.
         */
    status=tt_message_status(m)

    /* set string based on type of error */
    if (tt_is_error(status)) {
        if (tt_is_sb_ide_error(status)) {
            string = tt_message_status_string(m);
        } else {
            /* cast really not needed */
            string = tt_status_message((Tt_status)status);
        }
    }

    /* run code for request answer */
    if (!tt_is_error(status)) {
        /* do normal stuff here with, say, a message argument */
        do_stuff(tt_message_arg_val(m, 0));
    } else {
        /* process error here */
        print_error_to_footer(string);
    }

    /* done */
    tt_message_destroy(m);
    return TT_CALLBACK_PROCESSED;
}
```

Data Exchange

Generate Program Design Information

The request sent by CDE to the SourceBrowser to ask it to generate the program design information in IDF format for the database in the specified directory is the `IDE_gen_design` message. The name of the IDF file is returned. It is stored in the specified directory. If the boolean flag is zero, then the IDF file will contain information on source files residing in the database directory. If the boolean flag is not zero, then the IDF file will also contain information on source files belonging to the program, but residing in other directories.

```
(Tt_message
   (Tt_class TT_REQUEST) (Tt_op "IDE_gen_design")
   (Tt_args (TT_IN "directory" directory)
      (TT_IN "boolean" lib)
      (TT_OUT "string" idf_file)))
```

directory	SourceBrowser database in this directory
lib	One means do this directory only; zero means do other directories also
idf_file	File containing data in IDF format

Error Codes:

`TT_ERR_NO_MATCH`	Message not handled (SourceBrowser not running)
`SB_IDE_ERR_NODIR`	Directory not valid (SourceBrowser cannot open that directory)
`SB_IDE_ERR_NODB`	No database found (SourceBrowser has no database in that directory)
`SB_IDE_ERR_NOWRITE`	Directory not writable (SourceBrowser cannot write in that directory)
`SB_IDE_ERR_NODUMP`	Unable to execute `sbidfdump` program

Generate Design Information

The `IDE_sync` request is sent by CDE to the SourceBrowser to ask it to generate design information in IDF format from the specified source file in the specified directory. The name of the IDF file is returned. It is stored in the specified directory.

```
(Tt_message
   (Tt_class TT_REQUEST) (Tt_op "IDE_sync")
   (Tt_args (TT_IN "string" directory)
      (TT_IN "boolean" lib)
      (TT_OUT "string" idf_file)
      (TT_IN "file" source_file)))
```

directory	SourceBrowser database in this directory
lib	This argument ignored
idf_file	File containing data in IDF format
source_file	Source file to reverse engineer

Error Codes:

`TT_ERR_NO_MATCH`	Message not handled (SourceBrowser not running)
`SB_IDE_ERR_NODIR`	Directory not valid (SourceBrowser cannot open that directory)
`SB_IDE_ERR_NODB`	No database found (SourceBrowser has no database in that directory)
`SB_IDE_ERR_NOWRITE`	Directory not writable (SourceBrowser cannot write in that directory)
`SB_IDE_ERR_NOFILE`	Source file not found
`SB_IDE_ERR_NODUMP`	Unable to execute `sbidfdump` program

IDF File Generated

The `IDF_generated` notice is sent by the `sbidfdump` program to the SourceBrowser when it has completed the generation of an IDF file. When the SourceBrowser receives an `IDE_gen_design` or `IDE_sync` request (defined above), it forks the `sbidfdump` program, waits to receive the `IDF_generated` notice from `sbidfdump`, and then responds to the original IDE request.

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "IDF_generated")
   (Tt_args (TT_IN "string" idf_file_name)
      (TT_IN "string" idf_file_type)))
```

idf_file_name	File containing data in IDF format or error message
idf_file_type	"`IDF_FILE`" or "`SYNC_FILE`" or "`ERROR`"

Error Codes:

`SB_IDE_ERR_NODIR`	Directory not valid (`sbidfdump` cannot open that directory)
`SB_IDE_ERR_NODB`	No database found (SourceBrowser has no database in that directory)
`SB_IDE_ERR_NOWRITE`	Directory not writable (`sbidfdump` cannot write in that directory)
`SB_IDE_ERR_NOFILE`	Source file not found

Display Synchronization

Display Source Code

The message sent by CDE to the SourceBrowser to ask it to display the source code for an object, in the specified file, is the `SPRO_Visit_Object` message. If the SourceBrowser's current directory is different, it changes its directory (which will cause its current queries, graphs, and browsed classes to be lost). The SourceBrowser does a query for the object of the specified name, and if it is found in the specified file, the source code is displayed.

```
(Tt_message
   (Tt_class TT_REQUEST) (Tt_op "SPRO_Visit_Object")
   (Tt_args (TT_IN "string" directory)
      (TT_IN "string" name)
      (TT_IN "string" file)))
```

directory	SourceBrowser database set to this directory
name	Name of object. For a “datamember”, the format of the name must be class::member.
file	File containing source for the object

Error Codes:

`TT_ERR_NO_MATCH`	Message not handled (SourceBrowser not running)
`SB_IDE_ERR_NODIR`	Directory not valid (SourceBrowser cannot open that directory)
`SB_IDE_ERR_NODB`	No database found (SourceBrowser has no database in that directory)
`SB_IDE_ERR_INVOBJ`	Object_type invalid (not one of those listed above)
`SB_IDE_ERR_NOSYMD`	Name not found in database (either directory is wrong, or it’s just not there, or it’s formatted incorrectly)
`SB_IDE_ERR_NOSYMF`	Name not found in file (name found, but definition is not in the specified file)

Display Design Diagram

This message is sent by the SourceBrowser when it wants the `cnav` tool to start up the appropriate IDE design diagram editor to display the design diagram for the specified object of the specified name.

```
(Tt_message
   (Tt_class TT_REQUEST) (Tt_op "IDE_browse")
   (Tt_args (TT_IN "string" directory)
      (TT_IN "string" name)))
```

directory	Source file in this directory. NOTE: This may NOT be the database directory.
name	Name of object. For a "datamember", the format of the name must be class::member.

Error Codes:

`TT_ERR_NO_MATCH`	Message not handled (`cnav` not running)
`SB_IDE_ERR_NODIR`	Directory not valid (`cnav` cannot open that directory)
`SB_IDE_ERR_NODB`	No database found (`cnav` has no database in that directory)
`SB_IDE_ERR_INVOBJ`	Object_type invalid (not one of those listed above)
`SB_IDE_ERR_NOSYMD`	Name not found in database (either directory is wrong, or it's just not there, or it's formatted incorrectly)

Debugger Integration

The Debugging messages are primarily notices which keep you informed of state between dbx and the Debugger. If your application can benefit from the dynamic state of the Debugger, then control integration can be accommodated through the notification of dbx events. For instance, you might have a control flow diagram which can follow alongside the Debugger by being aware of the program state.

Data integration is accomplished through the Debugger's stabs interface. stabs is a comprehensive and detailed set of information about the structure of your program. It includes program information such as block delimiters, include files, source and directory file names, and segment information. For symbols, it lists type, global, local, register, function, parameter, and other program details for C, C++, Fortran, and Pascal. This information is much more complex than the SourceBrowser database information, but if your application is source-detail oriented, then this will give you all the information you need. The *SPARCworks 2.0 Debugger Interface* manual describes the stabs format and is available through SunPro's Partners Integration Program.

The Debugging messages include those associated with properties, dbx events, and the Debugger View Port.

The messages associated with properties include SPRO_SET_PROP, SPRO_GET_PROP, SPRO_GET_PROPS, and SPRO_SEND_PROP.

The messages associated with dbx events include SPRO_SE_DEBUGGED, SPRO_SE_DROPPED, SPRO_SE_STARTED, SPRO_SE_REVIVED, SPRO_SE_ATTACHED, SPRO_SE_GONE, SPRO_SE_THREAD, SPRO_TE_STOPPED, SPRO_TE_TRACED, SPRO_TE_CONTINUED, SPRO_TE_STEPPED, SPRO_TE_VISIT, SPRO_TE_LOCATION, SPRO_TE_EVENTSET, SPRO_TE_EVENTDEL, and SPRO_SE_FIXED.

Debugging functionality is divided between the debugging engine (dbx) and the Debugger. Each Debugger will display a single instance of a Debugger View Port (DVP) tool. A DVP is associated with a program/process/thread combination. The association can be changed using the Process/Thread Inspector which is part of the Debugger.

There will be one instance of the dbx debugging engine running for each target process being debugged with each dbx is running a single ksh command interpreter. Thus changing the process associated with a DVP implies a changing the dbx engine to which the DVP is sending commands. Each dbx engine is associated with a current thread. Thus changing the thread associated with a DVP implies changing the thread context of a dbx engine. In this implementation there is never more than one DVP or Interpreter.

ID's

The protocol uses unique ID's to identify objects and events. The ID types are

- *Debuggee_ID*: an ID for <*executable, engine*> generated when a program is loaded.
- *Process_ID*: an ID for a Unix process (or the pseudo process represented by a core file).
- *Thread_ID*: an ID for a thread within a process.
- *Trap_ID*: an ID for a breakpoint
- *Context_ID*: an ID for a DVP

A *Context ID* is explicitly included as a parameter in almost all messages in order to allow for future releases which may have multiple DVP's and multiple command interpreters.

Most ID's are constructed by concatenating a tool's ToolTalk process ID with the value of a counter maintained by that process. The form used is produced by:

```
sprintf(buf, "%d %s %s", counter++, type_name, tt_procid);
```

The ID for a new process is built by:

```
sprintf(buf, "0 %d Process_ID %s", counter++, tt_procid);
```

The thread ID for the initial thread is the same as the process ID. The thread IDs for subsequent threads are formed by replacing the leading zero with the thread index.

Messages with:

```
(Tt_address TT_HANDLER)(Tt_handler procid)
```

procid	TT process ID of intended recipient

are delivered only to the designated process.

The Debugger, `dbx`, pio, and VDI used the second form for much of their private communication. The Debugger passes its process ID to tools that it starts. They identify themselves with `SPRO_ATTACH`, `SPRO_PIO_ATTACH`, or `SPRO_VDI_ATTACH` messages back to the Debugger which get their process ID from the sender field.

Properties

The processes used for debugging exchange information as <*name, value*> pairs using the messages `SPRO_SET_PROP`, `SPRO_GET_PROP`, `SPRO_SEND_PROPS`, and `PRO_GET_PROPS`.

Set Properties

```
(Tt_message
   (Tt_class TT_REQUEST) (Tt_op "SPRO_SET_PROP")
   (Tt_address TT_HANDLER) (Tt_handler procid)
   (Tt_args (TT_IN "Context_ID" clique_context)
      (TT_IN "string" property_name)
      (TT_IN "string" property_value)))
```

procid	TT process ID of intended recipient
clique_context	The context as defined by the Debugger
property_name	The property to be retrieved (see Table 2-1 for details)
property_value	The value to which *property_name* is to be set

Get Properties

```
(Tt_message
   (Tt_class TT_REQUEST) (Tt_op "SPRO_GET_PROP")
   (Tt_address TT_HANDLER) (Tt_handler procid)
   (Tt_args (TT_IN "Context_ID" clique_context)
      (TT_IN "string" property_name)
      (TT_OUT "string" property_value)))
```

procid	TT process ID of intended recipient
clique_context	The context as defined by the Debugger

property_name	The property to be retrieved
property_value	The value of *property_name*

The properties which can be discussed with the `SET_PROP` and `GET_PROPS` requests are:

Table 2-1: SET_PROP and GET_PROP properties

Name	Value	Request Sent From	Request Sent To
`program_io_add`	The process ID of the process running the input/output window. This is sent by the Debugger after it starts the window.	Debugger	`dbx`
`vdi_proc_add`	The process ID of the process running the Visual Data Inspector window. This is sent by the Debugger after it starts the window.	Debugger	`dbx`
`toolenv options`[a]	The value is the new value for the tool environment variable as documented for the `dbx toolenv` command.	`dbx`	Debugger
`dbxenv` options[b]	The name can be any of the `dbxenv` options. See the Debugger documentation or issue the `dbx` command `dbxenv` without any options to see the list. The value is the new value of the `dbxenv` option, see the Debugger documentation for valid options and their meanings.	anyone	`dbx`
`setenv`	Arguments formatted for `putenv (3)` describing changes in the environment. This is needed by the process input/output window.	`dbx`	pio
`program_name`	The name of the program being debugged.	Debugger	pio

a. toolenv options are *font*, *width*, *srclines*, *cmdlines*, *displines*, *topmargin*, and *botmargin*
b. dbxenv options are *case*, *shortfname*, *makeargs*, *speed*, *stringlen*, *autoflush*, *logfile*, *stepevents*, *stackmaxsize*, *stackverbose*, *lookaside*, *warnings*, *follow_fork_mode*, *follow_fork_inherit*, *compress_stabs*, *aout_cache_size*, *lo_cache*, *mt_watchpoints*, *func_overload*, *pop_autodestruct*, *setpgrp*, *dynamic_type*, *errlogfile*, *autocontinue*, *error_limit* and *autosuppress*

Send Properties

For state changes which are actively shared between the engine and some other process, the following notification is used:

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "SPRO_SEND_PROP")
   [(Tt_address TT_HANDLER) (Tt_handler procid)]
   (Tt_args (TT_IN Context_ID clique_context)
      (TT_IN "string" property_name)
      {(TT_IN "string" property_value)}+))
```

procid	TT process ID of intended recipient
clique_context	The context as defined by the Debugger
property_name	The property which is being broadcast
property_value	The value of *property_name*

The supported properties are listed in Table 2-2:

Table 2-2: SEND_PROP Properties

Name	Value	Notice Sent From
`use`	The colon-separated list of directories used for the `dbx use` command	`dbx`
`dbxenv` options[a]	The name can be any of the `dbxenv` options. See the Debugger documentation or issue the `dbx` command `dbxenv` without any options to see the list. The value is the new value of the `dbxenv` option, see the Debugger documentation for valid options and their meanings.	`dbx`
`cwd`	The current working directory of `dbx`	`dbx`
`stack_filters`	This is a list of filter pairs where the first string of each pair is the filter number and the second string of each pair is the filter string itself	`dbx` (only used by Debugger)

Table 2-2: SEND_PROP Properties

Name	Value	Notice Sent From
`sunperf`	A Collector parameter followed by a space, followed by its new value	`dbx` (only used by Debugger)

a. dbxenv options are *case*, *shortfname*, *makeargs*, *speed*, *stringlen*, *autoflush*, *logfile*, *stepevents*, *stackmaxsize*, *stackverbose*, *lookaside*, *warnings*, *follow_fork_mode*, *follow_fork_inherit*, *compress_stabs*, *aout_cache_size*, *lo_cache*, *mt_watchpoints*, *func_overload*, *pop_autodestruct*, *setpgrp*, *dynamic_type*, *errlogfile*, *autocontinue*, *error_limit* and *autosuppress*

Get All Properties

The `dbxenv` and use list properties can be gotten with the single request:

```
(Tt_message
   (Tt_class TT_REQUEST) (Tt_op "SPRO_GET_PROPS")
   (Tt_address TT_HANDLER) (Tt_handler procid)
   (Tt_args (TT_IN "Context_ID" clique_context)
      (TT_OUT "string" property_list)))
```

procid	TT process ID of intended recipient
clique_context	The context as defined by the Debugger
property_list	A semicolon separated list of property strings of the form taken by `putenv(3)` followed by the "use" list of `dbx`. (Refer to the Debugger documentation for more information on the `use` command)

dbx Events

When a `dbx` engine is running under the SPARCworks Debugger, it notifies the Debugger and other interested clients of engine status changes via ToolTalk notifications.

Program Change

If a `dbx` engine loads a new program it sends:

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "SPRO_SE_DEBUGGED")
   (Tt_args (TT_IN "Context_ID" clique_context)
      (TT_IN "Debuggee_ID" debuggee_id)
      (TT_IN "string" current_dir)
      (TT_IN "string" filename)))
```

clique_context	A clique_context that has been sent from the Debugger
debuggee_id	A new Debuggee ID invented by the engine
current_dir	The current working directory
filename	The filename of the program that is being debugged

Engine left without Program

If an engine is left without an associated program, the engine sends:

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "SPRO_SE_DROPPED")
   (Tt_args (TT_IN "Debuggee_ID" debuggee_id)))
```

debuggee_id	The ID of the program that is no longer being debugged

Process Change

When a process is started due to user action, `dbx` sends:

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "SPRO_SE_STARTED")
   (Tt_args (TT_IN "Context_ID" clique_context)
      (TT_IN "Process_ID" process_id)
      (TT_IN "string" host)
      (TT_IN "int" pid)))
```

clique_context	A clique_context that has been sent from the Debugger
process_id	A unique ID for the new process
host	The host on which the process is running

pid	The PID of the process

Process Resumed from Core File

If a process is resumed from a core file the engine sends:

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "SPRO_SE_REVIVED")
   (Tt_args (TT_IN "Context_ID" clique_context)
      (TT_IN "Process_ID" process_id)
      (TT_IN "string" corefile)))
```

clique_context	A clique_context that has been sent from the Debugger
process_id	A unique ID for the new process
corefile	The path to the corefile that is now being debugged

Attachment to Process Signalled

Attachment to a process is signalled by:

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "SPRO_SE_ATTACHED")
   (Tt_args (TT_IN "Context_ID" clique_context)
      (TT_IN "Process_ID" process_id)
      (TT_IN "int" pid)))
```

clique_context	A clique_context that has been sent from the Debugger
process_id	A unique ID for the new process
pid	The PID of the process that is now being debugged

Engine Loses Process

If an engine loses its process, it sends:

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "SPRO_SE_GONE")
   (Tt_args (TT_IN "Process_ID" process_id)))
```

process_id	The unique ID for the process

Thread Change

The following message is generated when dbx switches threads.

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "SPRO_SE_THREAD")
   (Tt_args (TT_IN "Context_ID" clique_context)
      (TT_IN "Thread_ID" thread)))
```

clique_context	A clique_context that has been sent from the Debugger
thread	The thread ID of the thread that is now being debugged

Target Process Events

When a dbx engine is running under the SPARCworks Debugger, it notifies the Debugger and other interested clients of target process status changes via ToolTalk notifications. Many of these messages may occur spontaneously without user action.

Thread State

When a thread stops, the engine sends:

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "SPRO_TE_STOPPED")
   (Tt_args (TT_IN "Thread_ID" stopped_thread)
      (TT_IN "string" visit_source)
      (TT_IN "string" visit_func)
      (TT_IN "int" visit_srcline)
```

```
      (TT_IN "string" home_source)
      (TT_IN "string" home_func)
      (TT_IN "int" home_srcline)))
```

stopped_thread	The thread ID of the thread that has stopped
visit_source	The file of the closest frame up the call stack for which `dbx` has debugging information
visit_func	The function of the closest frame up the call stack for which `dbx` has debugging information
visit_srcline	The line of the closest frame up the call stack for which `dbx` has debugging information
home_source	The file of the frame at the top of the stack
home_func	The function of the frame at the top of the stack
home_srcline	The line of the frame at the top of the stack

The information for home and visiting locations control the solid and hollow source arrows in the Debugger. The home location is denoted by the solid arrow and the visiting location by the hollow arrow. If the home and visiting locations are the same then only the solid arrow is shown.

When a MT program stops, this message is sent for the first thread to hit a breakpoint (or otherwise stop) and that thread becomes the current thread. All other threads stop without this message being generated.

Trace Point Signalled

A trace point is signalled by:

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "SPRO_TE_TRACED")
   (Tt_args (TT_IN "Thread_ID" traced_thread)
      (TT_IN "string" visit_source)
      (TT_IN "string" visit_func)
      (TT_IN "int" visit_srcline)
```

```
(TT_IN "string" home_source)
(TT_IN "string" home_func)
(TT_IN "int" home_srcline)))
```

traced_thread	The thread ID of the thread that is being traced
visit_source	The file of the closest frame up the call stack for which `dbx` has debugging information
visit_func	The function of the closest frame up the call stack for which `dbx` has debugging information
visit_srcline	The line of the closest frame up the call stack for which `dbx` has debugging information
home_source	The file of the frame at the top of the stack
home_func	The function of the frame at the top of the stack
home_srcline	The line of the frame at the top of the stack

Execution Continued

Resumption of execution is announced by either `SPRO_TE_CONTINUED` or `SPRO_TE_STEPPED`.

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "SPRO_TE_CONTINUED")
   (Tt_args (TT_IN "Thread_ID" thread)
      (TT_IN "string" command_name)))
```

thread	The thread ID of the current thread

Execution Resumed

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "SPRO_TE_STEPPED")
   (Tt_args (TT_IN "Thread_ID" thread)))
```

thread	The thread ID of the current thread

Shift in Stack Frame Context

A shift in stack frame context is signalled by:

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "SPRO_TE_VISIT")
   (Tt_args (TT_IN "Thread_ID" thread)
      (TT_IN "string" visit_source)
      (TT_IN "string" visit_func)
      (TT_IN "int" visit_srcline)))
```

thread	The thread ID of the thread that is being visited
visit_source	The file of the closest frame up the call stack for which `dbx` has debugging information
visit_func	The function of the closest frame up the call stack for which `dbx` has debugging information
visit_srcline	The line of the closest frame up the call stack for which `dbx` has debugging information

Potential Shift in Current Source Location

When the location at which a thread stopped is returned to (as after a manual procedure call) or when the current thread is changed, the potential shift in current source location is signalled using a `SPRO_TE_LOCATION` message. This same notification is used to announce the location of the first line of `main()` when a program is first loaded.

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "SPRO_TE_LOCATION")
   (Tt_args (TT_IN "Thread_ID" thread)
      (TT_IN "string" visit_source)
      (TT_IN "string" visit_func)
      (TT_IN "int" visit_srcline)
      (TT_IN "string" home_source)
      (TT_IN "string" home_func)
      (TT_IN "int" home_srcline)))
```

thread	The thread ID of the thread that is being manipulated or null if finding the first line of main at program load time

visit_source	The file of the closest frame up the call stack for which dbx has debugging information
visit_func	The function of the closest frame up the call stack for which dbx has debugging information
visit_srcline	The line of the closest frame up the call stack for which dbx has debugging information
home_source	The file of the frame at the top of the stack or null if finding the first line of main at program load time
home_func	The function of the frame at the top of the stack or null if finding the first line of main at program load time
home_srcline	The line of the frame at the top of the stack or null if finding the first line of main at program load time

Event Status

The setting and deletion of an event specification are signalled by:

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "SPRO_TE_EVENTSET")
   (Tt_args (TT_IN "Thread_ID" thread)
      (TT_IN "Trap_ID" trap_id)
      (TT_IN "string" event_type)
      (TT_IN "string" symbol)
      (TT_IN "string" file)
      (TT_IN "int" lineno)
      (TT_IN "string" condition)
      (TT_IN "string" action)))
```

thread	This parameter is unused
trap_id	The trap ID of this event
event_type	This parameter is unused
symbol	This parameter is unused
file	The file name where this event was set
lineno	The line number where this event was set

condition	This parameter is unused
action	This parameter is unused

Event Specification Deleted

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "SPRO_TE_EVENTDEL")
   (Tt_args (TT_IN "Trap_ID" trap_id)
      (TT_IN "string" event_type)
      (TT_IN "string" symbol)
      (TT_IN "string" file)
      (TT_IN "int" lineno)
      (TT_IN "string" condition)
      (TT_IN "string" action)))
```

trap_id	The trap ID of this event
event_type	This parameter is unused
symbol	This parameter is unused
file	The file name where this event was set
lineno	The line number where this event was set
condition	This parameter is unused
action	This parameter is unused

As implemented, only the trap_id, file and lineno parameters have useful values.

Fix Command Completed

The engine announces completion of the fix command by sending:

```
(Tt_message
   (Tt_class TT_NOTICE) (Tt_op "SPRO_SE_FIXED")
   (Tt_args (TT_IN "Context_ID" clique_context)
      (TT_IN "string" filename)))
```

clique_context	A clique_context that has been sent from the Debugger
filename	The file that has been fixed

Debugger View Port

Source Display

`dbx` controls the DVP source display using the message:

```
(Tt_message
   (Tt_class TT_REQUEST) (Tt_op "SPRO_Visit_File")
   (Tt_args (TT_IN "Context_ID" clique_context)
      (TT_IN "string" file)
      (TT_IN "int" lineno)
      (TT_IN "int" centered)))
```

clique_context	A clique_context that has been sent from the Debugger
file	The file that should be displayed
lineno	The line number that should be displayed
centered	If zero, then the line number should appear at the top of the display. If nonzero, then it should be centered.

MakeTool Integration

Control integration in MakeTool is accomplished with two ToolTalk messages. One of these messages, `MAKETOOL_EXEC`, is for compilation control, while the other, `MAKETOOL_COMPILE`, is for task control. Both messages utilize MakeTool's flexible functionality.

The `MAKETOOL_EXEC` message is used specifically for requesting a build using the `make` command. Any tool which allows the user to modify code can also give the user an integrated way of recompiling that code through this message.

The `MAKETOOL_COMPILE` message allows you to generate any arbitrary command, such as a compile, to be executed through MakeTool. For example, you could execute simple commands such as "`cc -g -o hello hello.c`" or "`ls -alF`." Since the command does not have to be a compilation command, you could also have MakeTool execute something as complex as a daily testing script which may take hours to complete. MakeTool properties may be set up to send mail to a user upon completion or failure of the executed task. MakeTool's animated icon keeps the user informed as to the status of the task without requiring an open shell window.

Beginning a Build

This is the request to begin a build.

```
(Tt_message
   (Tt_class TT_REQUEST) (Tt_op "MAKETOOL_EXEC")
   (Tt_file working_directory)
   (Tt_args (TT_IN "string" environment)
      (TT_IN "string" arguments)))
```

working_directory	The directory in which the build should occur
environment	A newline-separated list of strings of the form taken by `putenv` that specify the environment in which the process will run
arguments	A space-separated list of arguments to be passed to `make`

Executing a Command

This is a request to synchronously execute an arbitrary command, usually a compile.

```
(Tt_message
   (Tt_class TT_REQUEST) (tt_op "MAKETOOL_COMPILE")
   (Tt_file cwd)
   (Tt_args (TT_IN "string" environment)
      (TT_IN "string" arguments)
      (TT_OUT "int" status)))
```

cwd	The directory in which to execute the command
environment	The shell environment of the tool that sent the message
arguments	The entire command line, e.g., `cc -o foo foo.c`
status	The results of the command

FileMerge Integration

Your application can take advantage of control integration with FileMerge with message requests designed to allow you to load and unload files into and out of FileMerge as well as save the merged output from FileMerge. If your application handles multiple versions of files and then depends on one merged version to be the final copy, then use FileMerge to help resolve the differences between the files.

FileMerge notices include those which change a FileMerge message and indicate a FileMerge started, loaded, unloaded, saved, and quitted.

FileMerge Start-up

Most FileMerge communication is point-to-point based. Once a ToolTalk connection is established, the client application must have the FileMerge process ID in order for FileMerge to respond to its ToolTalk messages.

FileMerge can be started by a client application and maintain ToolTalk communication using the following steps:

- Get the process ID for the application
- `fork(2)` and `exec(2)` FileMerge with the appropriate option
- Retrieve the FileMerge process ID (*fm_procid*) from the return ToolTalk message

Getting the process ID

The process ID for the client application is returned when `tt_open()` is called. See the section, "Open Communication and Obtain File Descriptor," in Chapter 2, "The ToolTalk Service," for more information on `tt_open()`.

Fork and Exec FileMerge

To completely control FileMerge, an application must start its own FileMerge. `fork` the client application, then `exec filemerge` with the `-T` *my_procid* option using the previously obtained *my_procid*. When FileMerge is started and sees the `-T` option, it will send a `SunPro_FileMerge_Started` ToolTalk message back to the client application using the *my_procid* argument.

Retrieving the FileMerge Process ID

The FileMerge process ID (*fm_procid*) is a char * which represents a handle to the new FileMerge. This handle should be used whenever ToolTalk messages are sent from the client application. When the client application receives the initial ToolTalk message from FileMerge, *fm_procid* can be extracted by doing the following:

```
char            *fm_procid;   /* FileMerge handle */
Tt_message      msg_in;       /* actual TT message */
char            *op;          /* TT message name */

msg_in = tt_message_receive();/* Get the message from TT */
op = tt_message_op(msg_in);
   /* Get the name of the message */

/*
 * If this is the initial message from the FileMerge we started,
 * get fm_procid (which is the only argument passed in this
 * message)
 */
if (strcmp(op, "SunPro_FileMerge_Started") == 0) {
   fm_procid = tt_message_sender(msg_in);
```

All FileMerge messages with the exception of `SunPro_FileMerge_Load` and `SunPro_FileMerge_Load_RO` must specify *fm_procid* in order to perform their respective operations. Both load messages may be sent to any existing FileMerge. However, unloading, saving and quitting must be targeted toward a specific FileMerge identified by *fm_procid*.

FileMerge Requests

Loading Files into FileMerge

To load a set of files into a FileMerge, send either the `SunPro_FileMerge_Load` or `SunPro_FileMerge_Load_RO` message:

```
   (Tt_message
      (Tt_class TT_REQUEST)(Tt_op
["SunPro_FileMerge_Load"|"SunPro_FileMerge_Load_RO"])
      [(Tt_address TT_PROCEDURE)|
```

```
(Tt_address TT_HANDLER) (Tt_handler fm_procid)]
(Tt_args (TT_IN "integer" force_load)
  (TT_IN "integer" ignoreblanks)
  (TT_IN "string" ancestorfile)
  (TT_IN "string" leftlabel)
  (TT_IN "string" rightlabel)
  (TT_IN "string" listfile)
  (TT_IN "string" leftfile)
  (TT_IN "string" rightfile)
  (TT_INOUT "string" outputfile)
  (TT_IN "string" outputlabel)
  (TT_IN "integer" save_button)
  (TT_IN "integer" auto_save)))
```

fm_procid	FileMerge process ID
force_load	If nonzero, and if `TT_HANDLER` and *fm_procid* are specified, will force a load even if there are already loaded or unsaved files
ignoreblanks	Zero to make blanks significant; one to make them insignificant in the diff
ancestorfile	The name of the common ancestor file or the null string
leftlabel	The title for the left window of FileMerge or the null string
rightlabel	The title for the right window of FileMerge or the null string
listfile	A listfile as documented for the `filemerge -l` option
leftfile	The file for the left window
rightfile	The file for the right window
outputfile	The output file or a null string (will default to `filemerge.out`)
outputlabel	The label for the output file window
save_button	If one, then there will be a Save button; if zero, then there will be no Save button (use with auto_save)

auto_save	If one, then the save will occur when all differences are resolved; if zero, then the user must manually save the merged output

If the Load request specifies a specific FileMerge process, then that FileMerge process will be used to attempt the load of the set of files (or list of files) given in the operation argument fields.

The specified FileMerge process may fail on the load in the following ways:

- For read-only FileMerge, there are currently files loaded
- For read/write FileMerge, the current set of loaded files have not been saved
- There are still more files in the list (-l) of files to be merged

If, on the other hand, no FileMerge process is specified, the ToolTalk service will deliver the Load message to the first FileMerge process it finds. Read-only Load messages will only be sent to read-only FileMerges, as will read/write FileMerges. For read-only FileMerges, if the first FileMerge process ToolTalk finds is “free” (no files loaded), then this FileMerge will be used to load in the specified files.

For read/write FileMerges, if the first FileMerge process ToolTalk finds is “free” (no files loaded, current files have been saved, no more files on list), then this FileMerge will be used to load in the specified files.

Whether the load succeeds or fails, FileMerge will send a reply to the client process using either the TT_HANDLED or the TT_FAILED message. If the load succeeds, FileMerge will send a second message, SunPro_FileMerge_Loaded, to the client process.

Unloading Files from FileMerge

To unload the files from FileMerge send:

```
(Tt_message
   (Tt_class TT_REQUEST)(Tt_op
"SunPro_FileMerge_Unload")
   (Tt_address TT_HANDLER)(Tt_handler fm_procid)
   (Tt_args (TT_IN "integer" force_unload)))
```

fm_procid	FileMerge process ID
force_unload	One to force the unload (even if the files are not written); zero otherwise

FileMerge (both read-only and read/write) does not register ToolTalk `Unload` message patterns. The `Unload` operation must specify a specific FileMerge process. The FileMerge process will unload the current set of files or list of files.

Whether the unload succeeds or fails, FileMerge will send a reply `TT_HANDLED` or `TT_FAILED` to the client process. If the unload succeeds, FileMerge will send another ToolTalk message, `SunPro_Filemerge_Unloaded`, to the client process.

Saving a FileMerge

A read/write FileMerge can have its files saved by sending it:

```
(Tt_message
   (Tt_class TT_REQUEST)(Tt_op "SunPro_FileMerge_Save")
   (Tt_address TT_HANDLER)(Tt_handler fm_procid)
   (Tt_args(TT_IN "integer" force_save)
      (TT_INOUT "string" outputfile)))
```

fm_procid	FileMerge process ID
force_save	If nonzero, then a `Save` will be forced
outputfile	If non-null, then this is where the current merged output will be stored. If null, then the filename where the merged output is stored is returned.

FileMerge (both read-only and read/write) does not register ToolTalk save message patterns. The `Save` operation must specify a specific FileMerge process.

Whether the `Save` succeeds or fails, FileMerge will reply to the client process with either the message `TT_HANDLED` or `TT_FAILED`. If the `Save` succeeds, FileMerge will send another ToolTalk message, `SunPro_FileMerge_Saved`, to the client process.

Forcing a FileMerge to Quit

To force the FileMerge to exit, send the `SunPro_FileMerge_Quit` message.

```
    (Tt_message
            (Tt_class  TT_REQUEST)(Tt_op
"SunPro_FileMerge_Quit")
       (Tt_address  TT_HANDLER)(Tt_handler  fm_procid)
       (Tt_args  (TT_IN  "integer"  force_quit)))
```

fm_procid	FileMerge process ID
force_quit	If one, then the FileMerge will exit without requiring a `Save` or `Unload`. If zero, then if a `Save` or `Unload` is needed, request will fail.

FileMerge (both read-only and read/write) does not register ToolTalk quit message patterns.

The `Quit` operation must specify a specific FileMerge process.

The `Quit` operation may fail in the following ways:

- For read-only FileMerge, there are currently files loaded
- For read/write FileMerge, the current set of loaded files have not been saved
- There are still more files in the list (`-l`) of files to be merged

If the `Quit` fails, FileMerge will reply to the client process with the message `TT_FAILED`.

If the `Quit` succeeds, FileMerge will reply to the client process with the message `TT_HANDLED` and a second message, `SunPro_FileMerge_Quitted`.

FileMerge Notices

Changing a FileMerge Message

The message in the lower left corner of the FileMerge window can be changed with the message:

```
(Tt_message
   (Tt_class TT_NOTICE)(Tt_op
"SunPro_FileMerge_Set_LFtr")
   (Tt_address TT_HANDLER)
   (Tt_handler fm_procid)
   (Tt_args (TT_IN "string" left_footer_string)))
```

fm_procid	FileMerge process ID
left_footer_ string	The new footer or a null string

Indicating FileMerge Started

When a FileMerge is started and has completed its initial start-up it will send the following message to the process that started it:

```
(Tt_message
   (Tt_class TT_NOTICE)(Tt_op
"SunPro_FileMerge_Started")
   (Tt_address TT_HANDLER)(Tt_handler fm_procid)
```

fm_procid	FileMerge process ID

Indicating FileMerge Loaded

When a FileMerge has successfully loaded a set of requested files it will send the following message to the process that requested the load:

```
(Tt_message
   (Tt_class TT_NOTICE)(Tt_op "SunPro_FileMerge_Loaded")
   (Tt_address TT_HANDLER)(Tt_handler fm_procid)
   (Tt_args (TT_IN "string" ancestorfile)
      (TT_IN "string" listfile)
```

```
            (TT_IN "string" leftfile)
            (TT_IN "string" rightfile)
            (TT_IN "string" outputfile)))
```

fm_procid	FileMerge process ID
ancestorfile	The name of the common ancestor file
listfile	A listfile as specified in the `Load` command
leftfile	The file for the left window
rightfile	The file for the right window
outputfile	The output file

Indicating FileMerge Unloaded

When an `Unload` has completed, the client process will be sent:

```
   (Tt_message
      (Tt_class TT_NOTICE)(Tt_op
"SunPro_FileMerge_Unloaded")
      (Tt_address TT_HANDLER)(Tt_handler fm_procid)
      (Tt_args (TT_IN "string" ancestorfile)
         (TT_IN "string" listfile)
         (TT_IN "string" leftfile)
         (TT_IN "string" rightfile)
         (TT_INOUT "string" outputfile)))
```

fm_procid	FileMerge process ID
ancestorfile	The name of the common ancestor file
listfile	A listfile as specified in the `load` command
leftfile	The file for the left window
rightfile	The file for the right window
outputfile	The output file

Indicating FileMerge Saved

When a `Save` is completed (either by user or programmatic request), the client will be sent:

```
(Tt_message
   (Tt_class TT_NOTICE)(Tt_op "SunPro_FileMerge_Saved")
   (Tt_address TT_HANDLER)(Tt_handler fm_procid)
   (Tt_args (TT_IN "string" ancestorfile)
      (TT_IN "string" leftfile)
      (TT_IN "string" rightfile)
      (TT_IN "string" outputfile)))
```

fm_procid	FileMerge process ID
ancestorfile	The name of the common ancestor file
leftfile	The file for the left window
rightfile	The file for the right window
outputfile	The output file

Indicating FileMerge Quitted

Just prior to quitting, FileMerge will send to its client:

```
(Tt_message
   (Tt_class TT_NOTICE)(Tt_op "SunPro_FileMerge_Quitted")
   (Tt_address TT_HANDLER)(Tt_handler fm_procid)
   (Tt_args (TT_IN "string" ancestorfile)
      (TT_IN "string" listfile)
      (TT_IN "string" leftfile)
      (TT_IN "string" rightfile)
      (TT_INOUT "string" outputfile)))
```

fm_procid	FileMerge process ID
ancestorfile	The name of the common ancestor file
listfile	A listfile as specified in the `Load` command
leftfile	The file for the left window

rightfile	The file for the right window
outputfile	The output file

Performance Analyzer Integration

Though the Performance Analyzer does not support any user-defined ToolTalk messages, data integration can still be accomplished. The `er_export(1)` utility can provide all the performance data in ASCII format so that it can be easily parsed and used by your application. Graphing tools can take advantage of this information and provide displays for the data that the Analyzer does not. The format of the `er_export` output is described in the manual *Performance Tuning Your Application*. This manual is part of the SPARCworks 3.0 documentation set.

ToolTalk Message Index

A

Table 1 lists ToolTalk messages alphabetically, with a reference to the page on which the message is described.

.

Table 1: ToolTalk Messages

Message Name	Message Description	Notice/ Request	Page No.
`departed`	SPARCworks Manager Integration message used to inform the SPARCworks Manager that a tool is about to quit.	Notice	39
`expose`	SPARCworks Manager Integration message sent to one or more tools running in the SPARCworks session to instruct the tool to expose a window.	Notice	38
`hide`	SPARCworks Manager Integration message sent to one or more tools running in the SPARCworks session to instruct the tool to hide a window.	Notice	38
`IDE_browse`	IDE Integration message sent by the SourceBrowser when it wants the `cnav` tool to start up the appropriate IDE design diagram editor	Request	49
`IDE_gen_design`	IDE Integration message sent by CDE to the SourceBrowser to ask it to generate the program design information in IDF format for the database in the specified directory.	Request	47
`IDE_quit`	IDE Integration message sent by CDE when it exits. The SourceBrowser receives, but then ignores, this message.	Notice	42
`IDE_sync`	IDE Integration message sent by CDE to the SourceBrowser to ask it to generate design information in IDF format from the specified source file in the specified directory.	Request	47
`IDF_generated`	IDE Integration message sent by the `sbidfdump` program to the SourceBrowser when it has completed the generation of an IDF file.	Notice	48
`MAKETOOL_EXEC`	MakeTool Integration message used to begin a build.	Request	66

Table 1: ToolTalk Messages

MAKETOOL_COMPILE	MakeTool Integration message used to synchronously execute an arbitrary command, usually a compile.	Request	66
PEI_CLOSE	SPARCworks Manager Integration message sent to one or more tools running in the SPARCworks session to instruct the tool to close a window.	Notice	39
PEI_OPEN	SPARCworks Manager Integration message sent to one or more tools running in the SPARCworks session to instruct the tool to open a window.	Notice	39
PEI_STARTED	SPARCworks Session Protocol message used by a SPARCworks tool to announce its existence when first started by SPARCworks Manager.	Request	37
quit	SPARCworks Manager Integration message sent to one or more tools running in the SPARCworks session to instruct the tool to quit a window.	Notice	39
SPRO_GET_PROP	Debugger Integration message used to get properties.	Request	53
SPRO_GET_PROPS	Debugger Integration message used to get all properties.	Request	56
SPRO_SBENG_START	SourceBrowser Integration message sent when SourceBrowser engine starts.	Notice	40
SPRO_SBENG_QUITS	SourceBrowser Integration message sent when sourceBrowser engine quits.	Notice	41
SPRO_SBENG_CURRENT _ELEMENT	SourceBrowser Integration message sent when the current match of the SourceBrowser source display changes.	Notice	41
SPRO_SE_ATTACHED	Debugger Integration message which signals attachment to a process.	Notice	58
SPRO_SE_DEBUGGED	Debugger Integration message sent when dbx engine loads a new program.	Notice	56

Table 1: ToolTalk Messages

SPRO_SE_DROPPED	Debugger Integration message sent by dbx if an engine is left without an associated program.	Notice	57
SPRO_SE_FIXED	Debugger Integration message used by dbx engine to announce completion of the fix command.	Notice	64
SPRO_SE_GONE	Debugger Integration message sent if a dbx engine loses its process.	Notice	58
SPRO_SE_REVIVED	Debugger Integration message sent by dbx engine if a process is resumed from a core file.	Notice	58
SPRO_SE_STARTED	Debugger Integration message sent by dbx when a process is started due to user action.	Notice	57
SPRO_SE_THREAD	Debugger Integration message generated when dbx switches threads.	Notice	59
SPRO_SEND_PROP	Debugger Integration message sent as notification of changes to the debugging engine.	Notice	55
SPRO_SET_PROP	Debugger Integration message used to set properties.	Request	53
SPRO_TE_CONTINUED	Debugger Integration message which indicates resumption of execution.	Notice	61
SPRO_TE_EVENTDEL	Debugger Integration message which signals the deletion of an event specification.	Notice	64
SPRO_TE_EVENTSET	Debugger Integration message which signals the setting of an event specification.	Notice	63
SPRO_TE_LOCATION	Debugger Integration message which signals a potential shift in current source location	Notice	62
SPRO_TE_STEPPED	Debugger Integration message which indicates resumption of execution by one program statement.	Notice	61
SPRO_TE_STOPPED	Debugger Integration message sent by dbx engine when a thread stops.	Notice	59
SPRO_TE_TRACED	Debugger Integration message which signals a trace point.	Notice	60

Table 1: ToolTalk Messages

`SPRO_TE_VISIT`	Debugger Integration message which signals a shift in stack frame context.	Notice	62
`SPRO_Visit_File`	Debugger Integration message used by `dbx` to control the DVP source display.	Request	65
`SPRO_Visit_Object`	IDE Integration message sent by CDE to the SourceBrowser to ask it to display the source code for an object, in the specified file.	Request	48
`SunPro_FileMerge_Load`	FileMerge Integration message used to load a set of files into a read/write FileMerge.	Request	69
`SunPro_FileMerge_Load_RO`	FileMerge Integration message used to load a set of files into a read-only FileMerge.	Request	69
`SunPro_FileMerge_Loaded`	FileMerge Integration message used when FileMerge has successfully loaded a set of requested files, to send message to the process that requested the load.	Notice	74
`SunPro_FileMerge_Quit`	FileMerge Integration message used to force a filemerge to exit.	Request	73
`SunPro_FileMerge_Quitted`	FileMerge Integration message used to indicate FileMerge just about to quit.	Notice	76
`SunPro_FileMerge_Save`	FileMerge Integration message used to save files in read/write FileMerge.	Request	72
`SunPro_FileMerge_Saved`	FileMerge Integration message used to indicate `save` completed (either by user or programmatic request).	Notice	76
`SunPro_FileMerge_Set_LFtr`	FileMerge Integration message used to change the message in the lower left hand corner of the FileMerge window	Notice	74
`SunPro_FileMerge_Started`	FileMerge Integration message used when FileMerge is started and has completed initial startup, to send message to the process that started it.	Notice	74
`SunPro_FileMerge_Unload`	FileMerge Integration message used to unload the files from a FileMerge.	Request	71

Table 1: ToolTalk Messages

SunPro_FileMerge_ Unloaded	FileMerge Integration message used to indicate unload completed.	Notice	75

SunPro
A Sun Microsystems, Inc. Business

2550 Garcia Avenue
Mountain View, CA 94043
415 960-1300
FAX 415 969-9131